JAVA

1. ♥ Java 概述 ♥
    - 1.1 什么是Java
    - 1.2 JDK 和 JRE, JVM的关系
    - 1.3 Java 三大版本
    - 1.6 JDK中的设计模式
    - 1.7 JDK 中常用的包
    - Java 与 C++ 的区别
2. ♥ Java 基础语法 ♥
    - 2.1 八大基本数据类
    - 2.2 Java Operation
    - 2.3 Java String
    - 2.4 流程控制语句
    - 2.5 static存在的主要意义
    - 2.6 final, static关键字
    - 2.7 面向对象和面向过程
    - 2.8 Java Access Modifier
    - 2.9 面向对象三大特性
    - 2.10 重写和重载
    - 2.11 Java 面向对象五大原则
    - 2.12 变量生命周期
    - 2.13 对象的生命周期
    - 2.14 JavaBean规范
    - 2.15 抽象类与接口
    - 2.16 变量与方法
    - 2.17 Object类的方法
    - 2.18 Java 异常处理机制
    - 2.19 Java 反射
    - 2.20 创建对象的五种方式
    - 2.21 Java注解
    - 2.22 Java Generic
    - 2.23 Java序列化
    - 2.24 IO
3. ♥ Java 集合框架 ♥
    - 3.1 Java Collections Framework
    - 3.2 JCF Various Interfaces
    - 3.3 List - ArrayList
    - 3.4 List - LinkedList
    - 3.5 List - Vector
    - 3.6 Set - HashMap & HashSet
    - 3.7 Set - ConcurrentHashMap
    - 3.8 Set - HashTable
    - 3.9 Set - TreeMap & TreeSet
    - 3.10 Set - LinkedHashMap & LinkedHashSet
    - 3.11 Queue - Deque & Array
    - 3.12 Queue - PriorityQueue
    - 3.12 总结
4. ♥ Java 多线程 ♥
    - 4.1 多线程概念
    - 4.2 为什么需要多线程
    - 4.3 线程状态转换

- 4.4 线程实现方式
- 4.5 线程中断方式
- 4.6 线程间通信
- 4.7 线程安全定义
- 4.8 并发三要素
- 4.9 Java怎么解决并发问题
- 4.10 Synchronized详解
- 4.11 Volatile 详解
- 4.12 线程安全的实现方法
- 4.13 JUC的结构
- 4.14 Java中的锁
- 4.15 JUC Locks LockSupport
- 4.16 JUC Locks AQS
- 4.17 JUC Locks AQS源码
- 4.18 JUC Locks ReentrantLock
- 4.19 JUC Locks ReadWriteLock
- 4.20 JUC ConcurrentHashMap
- 4.21 JUC ConcurrentLinkedQueue
- 4.22 JUC CopyOnWriteArrayList
- 4.23 JUC Java的阻塞队列
- 4.24 JUC ForkJoin
- 4.25 JUC Atomic CAS
- 4.26 JUC Atomic 原子操作类
- 4.27 JUC Tools CountDownLatch
- 4.28 JUC Tools CyclicBarrier
- 4.29 JUC Tools Semaphore
- 4.30 JUC Tools Exchanger
- 4.31 JUC 线程池 Executor
- 4.32 JUC 线程池 ThreadPoolExecutor
- 4.33 JUC 线程池 ScheduledThreadPoolExecutor
- 4.34 JUC 线程池 FutureTask

5. ♥ Java JVM ♥
   - 5.1 基本概念
   - 5.2 什么是字节码
   - 5.3 线程
   - 5.4 JVM 内存结构
   - 5.5 JVM 内存结构 程序计数器
   - 5.6 JVM 内存结构 虚拟机栈
   - 5.7 JVM 内存结构 本地方法栈
   - 5.8 JVM 内存结构 堆区
   - 5.9 逃逸分析
   - 5.10 JVM 内存结构 方法区
   - 5.11 GC 判断对象可回收
   - 5.12 GC 垃圾回收算法
   - 5.13 GC 引用类型
   - 5.14 GC 垃圾收集器
   - 5.15 JVM 类的生命周期
   - 5.16 类加载器
   - 5.17 JVM 类加载机制
   - 5.18 JVM 双亲委派
   - 缓存池 (buffer pool)
6. ♥ Java Design Patterns ♥

- 5.1 Design Patterns Concept
- 5.2 设计模式分类
- 5.3 UML图
- 5.4 软件设计原则
- 5.5 创建者模式
- 5.6 结构型模式
- 5.7 行为型模式

Reference

7. Java 基础
    - 001. What make Java Write Once and Run Anywhere
    - 002. Java 5 Design Principles
    - 003. Access modifiers in Java
    - 004. Purpose of static
    - 005. Purpose of final
    - 006. What is the constructor
    - 007. 重载和重写的区别
    - 008. 多态的理解
    - 009. & 和 && 区别
    - 010. == 和 equals 区别
    - 011. hashCode和equals 区别
    - 012. What is the abstraction
    - 013. 抽象类和接口的区别
    - 014. Java异常体系
    - 015. What is String Pool
    - 016. Two ways create String
    - 017. String, StringBuffer, StringBuilder
    - 018. Java Serialization
    - 019. Java Reflection
    - 020. Array 和 Collection 区别

    Comparable 和 Comparator区别
    - 021. ArrayList 和 Vector 区别
    - 022. ArrayList 和 LinkedList 区别
    - 023. Iterator 和 ListIterator 区别
    - 024. Iterator 和 Enumeration 区别
    - 025. 如何使ArrayList线程安全
    - 026. List和Set的区别
    - 027. HashMap Jdk1.7到1.8 区别
    - 028. HashMap的扩容机制
    - 029. HashMap的Put方法
    - 030. HashMap长度为2的幂次方原因
    - 031. HashMap 线程不安全
    - 032. 如何使HashMap线程安全
    - 033. HashMap 和 HashTable 区别
    - 034. HashSet 和 TreeSet 区别
    - 035. HashMap 和 TreeMap 区别
    - 036. 泛型中extends和super的区别
8. Java 并发
    - 001. Advantages of Multithreading
    - 002. 线程的生命周期
    - 003. 并发 并行 串行区别
    - 004. 并发的三大特性
    - 005. Thread 和 Runable 区别

- 006. 线程使用方式
- 007. sleep() wait() park()
  - sleep() 和 wait() 区别
  - sleep() 和 park() 的区别
  - wait() 和 park() 的区别
- 如果在wait()之前执行了notify()会怎样
- 如果在park()之前执行了unpark()会怎样?
- 008. join() 和 yield() 区别
- 009. synchronization 理解
- 010. 线程之间的互斥
- 011. 线程之间的协作
- 012. 线程安全的理解
- 013. 守护线程的理解
- 014. Java死锁如何避免
- 015. 线程池的用处以及参数
- 016. 线程池的底层工作原理
- 017. 线程池五种状态
- 018. 线程池中阻塞队列的作用
- 019. 线程池中线程复用原理
- 020. ExecutorService 理解
- 021. FutureTask 理解
- 022. 什么是原子操作
- 023. lock interface of JUC
- 024. ReentrantLock中的公平锁和非公平锁的底层实现
- 025 CountDownLatch和Semaphore的区别和底层原理
- 026. 对AQS的理解 AQS如何实现可重入锁
- 027. ThreadLocal详解

9. Java JVM
- 001. Understanding of Java virtual machine
- 002. 什么是字节码 采用字节码的好处
- 003. Types of memory areas are allocated by JVM
- 004. JVM中哪些是线程共享区
- 005. What is classloader
- 006. Java中有哪些类加载器
- 007. 说说类加载器双亲委派模型
- 008. 什么是Garbage collection
- 009. GC如何判断对象可以被回收
- 010. What is finalize()
- 011. JVM有哪些垃圾回收算法
- 012. 什么是STW
- 013. JVM有哪些垃圾回收器
- 014. 分代回收
- 015. CMS GC
- 016. G1 GC
- 017. 垃圾回收分为哪些阶段
- 018. 什么是三色标记
- 019. 项目如何排查JVM问题

10. Java 8 New Features
- 10.1 Lambda Expression
- 10.2 Default and Static Methods in Interfaces
- 10.3 Method References
- 10.4 Stream API

10.5 Optional Class
10.6 Date and Time API
10.7 Base64
10.8 Annotation
10.9 参数名字
10.10 调用JavaScript
10.11 并行数组

JAVA

1. ♥ Java 概述 ♥

1.1 什么是Java

- Java是一门面向对象编程语言，不仅吸收了C++语言的各种优点，还摒弃了 C++里难以理解的多继承、指针等概念，因此Java语言具有功能强大和简单易用两个特征
- Java语言作为静态面向对象编程语言的代表，极好地实现了面向对象理论，允许程序员以优雅的思维方式进行复杂的编程

1.2 JDK 和 JRE, JVM的关系

- JDK (Java SE Development Kit)，Java标准开发包，它提供了编译、运行Java程序所需的各种工具和资源, 也包括了JRE
- JRE (Java Runtime Environment)，Java运行环境，用于运行Java的字节码文件, 包括Java虚拟机和Java程序所需的核心类库等
    - 核心类库主要是java.lang包：包含了运行Java程序必不可少的系统类，如基本数 据类型、基本数学函数、字符串处理、线程、异常处理类等
    - 普通用户只需要安装JRE来运行Java程序，而程序开发者必须安装JDK来编译、调试程序
- JVM (Java Virtual Mechinal)，Java虚拟机，JRE的一部分, Java程序需要运行在虚拟机上，
    - 不同的平台有自己的虚拟机，因此Java语言可以实现跨平台
    - JVM不能单独搞定class的执行，解释class的时候JVM需要调用解释所需要的类库lib
    - 在JDK下面的的jre目录里面有两个文件夹bin和lib, bin里的就是jvm，lib中则是jvm工作所需要的类库
- JDK = JRE + DEV tools
- JRE = JVM + Java SE standard library
- 利用JDK（调用JAVA API）开发了JAVA程序后，通过JDK中的编译程序（javac）将java文件编译成JAVA字节码，在JRE上运行这些JAVA字节码，JVM解析这些字节码，映射到CPU指令集或OS的系统调用

1.3 Java 三大版本

- Java SE（J2SE，Java 2 Platform Standard Edition，标准版）
  - 它允许开发和部署在桌面、服务器、嵌入式环境和实时环境中使用的 Java 应用程序
  - Java SE 包含了支持 Java Web 服务开发的类，并为Java EE和Java ME提供基础。
- Java EE（J2EE，Java 2 Platform Enterprise Edition，企业版） Java EE 以前称为 J2EE
  - 企业版本帮助开发和部署可移植、健壮、可伸缩且安全的服务端Java 应用程序。Java EE 是在 Java SE 的基础上构建的，它提供 Web 服务、组件模型、 管理和通信 API，可以用来实现企业级的面向服务体系结构（service-oriented architecture，SOA）和 Web2.0应用程序。2018年2月，Eclipse 宣布正式将 JavaEE 更名 为 JakartaEE
- Java ME（J2ME，Java 2 Platform Micro Edition，微型版） Java ME 以前称为 J2ME
  - Java ME 为在移动设备和嵌入式设备（比如手机、PDA、电视 机顶盒和打印机）上运行的应用程序提供一个健壮且灵活的环境。Java ME 包括灵活的用 户界面、健壮的安全模型、许多内置的网络协议以及对可以动态下载的连网和离线应用程序 的丰富支持。基于 Java ME 规范的应用程序只需编写一次，就可以用于许多设备，而且可 以利用每个设备的本机功能。

1.6 JDK中的设计模式

- 适配器模式：Arrays.asList()，将数组转化为集合，底层还是数组，只是转换接口。不适用于基本数据类型（int,float等），可以get,set,不能调用修改集合大小的方法，如add，remove
- 代理模式：java.lang.reflect.Proxy，实现InvocationHandler接口，在invoke()前后加入代理
- 策略模式：java.util.Comparator函数接口的compare()方法。写好比较器然后传入sort()方法，执行不同的排序策略

1.7 JDK 中常用的包

- java.lang：这个是系统的基础类

  java.io：这里面是所有输入输出有关的类，比如文件操作等

  java.nio：为了完善 io 包中的功能，提高 io 包中性能而写的一个新包

  java.net：这里面是与网络有关的类

  java.util：这个是系统辅助类，特别是集合类

  java.sql：这个是数据库操作的类

Java 与 C++ 的区别

- Java 是纯粹的面向对象语言，所有的对象都继承自 java.lang.Object，C++ 为了兼容 C 即支持面向对象也支持面向过程。
- Java 通过虚拟机从而实现跨平台特性，但是 C++ 依赖于特定的平台。
- Java 没有指针，它的引用可以理解为安全指针，而 C++ 具有和 C 一样的指针。
- Java 支持自动垃圾回收，而 C++ 需要手动回收。
- Java 不支持多重继承，只能通过实现多个接口来达到相同目的，而 C++ 支持多重继承。

- Java 不支持操作符重载，虽然可以对两个 String 对象支持加法运算，但是这是语言内置支持的操作，不属于操作符重载，而 C++ 可以。
- Java 的 goto 是保留字，但是不可用，C++ 可以使用 goto。
- Java 不支持条件编译，C++ 通过 #ifdef #ifndef 等预处理命令从而实现条件编译

2. ♥ Java 基础语法 ♥

2.1 八大基本数据类

- 整数类型: byte(1个字节), short(2个字节), int(4个字节), long(8个字节)
- 浮点类型: float(4个字节, double(8个字节)
- 字符型: char(2个字节)
- 布尔型: boolean (4个字节)
- String 1.8是char[], 1.9 是byte[]
- boolean值在编译之后都使用Java虚拟机中的int数据类型来代替, 而boolean数组将会被编码成Java虚拟机的byte数组
  - 故boolean类型单独使用是4个字节，在数组中又是1个字节
- 两个Integer比较时
  - 当它们都在[-128,127]之间，使用"=="比较和equals比较返回的都是true；
  - 在此区间范围之外"=="比较为false，equals比较为true。原因是因为Integer中有一个内部类IntegerCache，当处于此区间的值都是相同的地址和值，因此此区间的==与equals都为true
- 数据类型
  - 八大基本数据类型算**基本类型 primitive datatype**
  - 另外还有**引用类型 reference datatype**，例如各种数组，String，创建的class
  - 和基本类型对应的叫**包装类型 wrapper classes**，Byte, Short, Integer, Long, Float, Double, Character, Boolean
    - The **autoboxing** is the process of converting primitive data type to the corresponding wrapper class object.
    - The **unboxing** is the process of converting wrapper class object to primitive data type.
  - 包装类型区别在于是对象，可以按照创建对象的流程创建

2.2 Java Operation

- float 与 double
  - 1.1 字面量属于 double 类型，不能直接将 1.1 直接赋值给 float 变量，因为这是向下转型。Java 不能隐式执行向下转型，因为这会使得精度降低。
  - 1.1f 字面量才是 float 类型

```
float f = 1.1f;
```

- Math.round
  - Math.round(11.5) 和 Math.round(-11.5) 等于多少?
- Math.round(11.5)的返回值是 12， Math.round(-11.5)的返回值是-11
  - 四舍五入的原理是在参数上加 0.5 然后进行下取整
- 隐式类型转换
  - 因为字面量 1 是 int 类型，它比 short 类型精度要高，因此不能隐式地将 int 类型下转型为 short 类型
  - 使用 += 运算符可以执行隐式类型转换

```
short s1 = 1;
// s1 = s1 + 1 会出错
s1 += 1;
```

- switch
  - 从 Java 7 开始，可以在 switch 条件判断语句中使用 String 对象

```
String s = "a";
switch (s) {
    case "a":
        System.out.println("aaa");
        break;
    case "b":
        System.out.println("bbb");
        break;
}
```

  - switch 不支持 long，是因为 switch 的设计初衷是对那些只有少数的几个值进行等值判断，如果值过于复杂，那么还是用 if 比较合适

2.3 Java String

- String 被声明为 final，因此它不可被继承。
- 内部使用 char 数组存储数据，该数组**被声明为 final**，这意味着不可被继承
- value 数组初始化之后就不能再引用其它数组。并且 String 内部没有改变 value 数组的方法，因此可以保证 String 不可变
- 不可变的好处
  - 可以缓存 hash 值, 用做 HashMap 的 key
  - String Pool 的需要, 如果一个 String 对象已经被创建过了，那么就会从 String Pool 中取得引用
  - 安全性, String 不可变性可以保证参数不可变
  - 线程安全, String 不可变性天生具备线程安全，可以在多个线程中安全地使用
- String.intern()
  - 使用 String.intern() 可以保证相同内容的字符串变量引用同一的内存对象

- s1 和 s2 采用 new String() 的方式新建了两个不同对象，结果不相等；使用双引号创建字符串会自动地将新建的对象放入 String Pool 中

```
String s1 = new String("aaa");
String s2 = new String("aaa");
// 一共会创建两个字符串对象
System.out.println(s1 == s2);           // false
```

```
String s3 = s1.intern();
System.out.println(s1.intern() == s3);  // true
```

```
String s4 = "bbb";
String s5 = "bbb";
// 会在堆内存中创建，堆中的bbb存的是pool中的指向
System.out.println(s4 == s5);  // true
```

- String Pool
  - String Pool 字符串常量池位于堆内存中，专门用来存储字符串常量，可以提高内存的使用率，避免开辟多块空间存储相同的字符串，在创建字符串时 JVM 会首先检查字符串常量池，如果该字符串已经存在池中，则返回它的引用，如果不存在，则实例化一个字符串放到池中，并返回其引用
  - 可以使用 String 的 intern() 方法在运行过程将字符串**添加到** String Pool 中。
  - 在 Java 7 之前，String Pool 被放在运行时常量池中，它属于永久代。而在 Java 7，String Pool 被移到**堆**中。这是因为永久代的空间有限，在大量使用字符串的场景下会导致 oom。
  - Java 8 之后取消永久代，类型信息、字段、方法、常量保存在本地内存的元空间，但字符串常量池、静态变量仍在堆中
  - 当一个字符串调用 intern() 方法时，如果 String Pool 中已经存在一个字符串和该字符串*值相等（使用 equals() 方法进行确定*），那么就会返回 String Pool 中字符串的引用；否则，就会在 String Pool 中添加一个新的字符串，并返回这个新字符串的引用
- 将字符串反转
  - 使用 StringBuilder 或者 stringBuffer 的 reverse() 方法

```
// StringBuffer reverse
StringBuffer stringBuffer = new StringBuffer();
stringBuffer.append("abcdefg");
System.out.println(stringBuffer. reverse()); // gfedcba
// StringBuilder reverse
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("abcdefg");
System.out.println(stringBuilder. reverse()); // gfedcba
```

- String 类的常用方法
  - indexOf()：返回指定字符的索引, charAt()：返回指定索引处的字符, replace()：字符串替换
  - length()：返回字符串长度
  - getBytes()：返回字符串的 byte 类型数组

- toLowerCase()：将字符串转成小写字母, toUpperCase()：将字符串转成大写字符
- substring()：截取字符串
- equals()：字符串比较
- split()：分割字符串，返回一个分割后的字符串数组, trim()：去除字符串两端空白

2.4 流程控制语句

- break 跳出此循环体，不再执行循环(结束当前的循环体)
- continue 跳出本次循环，继续执行下次循环(结束正在执行的循环 进入下一个循环条件)
- return 程序返回，不再执行下面的代码(结束当前的方法 直接返回)
- 在Java中，要想跳出多重循环，可以在外面的循环语句前定义一个标号，然后在里层循环体的代码中使用带有标号的break 语句，即可跳出外层循环

```
public static void main(String[] args) {
   ok:
   for (int i = 0; i < 10; i++) {
      for (int j = 0; j < 10; j++) {
         System.out.println("i=" + i + ",j=" + j);
         if (j == 5) {
            break ok;
}}}}
```

2.5 static存在的主要意义

- static的主要意义是在于创建独立于具体对象的域变量或者方法。以致于即使没有创建对象，也能使用属性和调用方法
- static关键字还有一个比较关键的作用就是用来形成静态代码块以优化程序性能。static块可以置于类中的任何地方，类中可以有多个static块
- **在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。**为什么说static块可以用来优化程序性能，是因为它的特性:只会在类加载的时候执行一次。因此，很多时候会将一些只需要进行一次的初始化操作都放在static代码块中进行
- 被static修饰的变量或者方法是独立于该类的任何对象，也就是说，这些变量和方法不属于任何一个实例对象，而是被类的实例对象所共享
- 在该类被第一次加载的时候，就会去加载被static修饰的部分，而且只在类第一次使用时加载并进行初始化，注意这是第一次用就要初始化，后面根据需要是可以再次赋值的
- static变量值在类加载的时候分配空间，以后创建类对象的时候不会重新分配。赋值的话，是可以任意赋值的
- 被static修饰的变量或者方法是优先于对象存在的，也就是说当一个类加载完毕之后，即便没有创建对象，也可以去访问
- static初始化可以不赋值
- static加载顺序简介
  - 先执行父类的静态代码块和静态变量初始化，并且静态代码块和静态变量的执行顺序只跟代码中出现的顺序有关

- 执行子类的静态代码块和静态变量初始化
- 执行父类的实例变量初始化
- 执行父类的构造函数
- 执行子类的实例变量初始化
- 执行子类的构造函数

2.6 final, static关键字

- final
  - **声明数据**为常量，可以是编译时常量，也可以是在运行时被初始化后不能被改变的常量。
    - 对于基本类型，final 使**数值不变**
    - 对于引用类型，final 使引用不变，也就不能引用其它对象，但是被引用的对象本身是可以修改的。final在修饰函数参数： 保证函数入参在函数体内不被改变
  - 声明方法**不能被子类重写**
    - private 方法隐式地被指定为 final，如果在子类中定义的方法和基类中的一个 private 方法签名相同，此时子类的方法不是重写基类方法，而是在子类中定义了一个新的方法。
  - 声明类**不允许被继承**
- static
  - **静态变量**：又称为类变量，这个变量属于类的，类所有的实例都共享静态变量，可以直接通过类名来访问它。静态变量在内存中只存在一份。
  - 对比实例变量：每创建一个实例就会产生一个实例变量，它与该实例同生共死。
  - 静态方法：在类加载的时候就存在了，它不依赖于任何实例。所以静态方法必须有实现，它不能是抽象方法。
  - 静态语句块：在类初始化时运行一次
  - 静态内部类：可以直接new一个外部类的内部类new MyClass().InnerClass()。非静态内部类依赖于外部类的实例，也就是说需要先创建外部类实例，才能用这个实例去创建非静态内部类
- final

2.7 面向对象和面向过程

- 面向过程：
  - 优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发，能是最重要的因素
  - 缺点：没有面向对象易维护、易复用、易扩展
- 面向对象：
  - 优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护
  - 缺点：性能比面向过程低

- 面向过程是具体化的，流程化的，解决一个问题，你需要一步一步的分析，一步 一步的实现。面向对象是模型化的，你只需抽象出一个类，这是一个封闭的盒子，在这里你拥有数据也拥有解决问题的方法。需要什么功能直接使用就可以了，不必去一步一步的实现，至于这个功能是如何实现的，管我们什么事？我们会用就可以了

2.8 Java Access Modifier

- Java 中有四个访问权限修饰符: public、protected、default 以及 private
- **public** 表示紧随其后的元素对任何人都是可用的。若是public class xxx，则其他包也可以访问该类。一个Java源文件中最多只能有一个public类，当有一个public类时，文件名必须与之一致，否则无法编译。如果源码中没有public类，则文件名与类中没有一致性要求
  - 类可见表示其它类可以用这个类创建实例对象
  - 成员可见表示其它类可以用这个类的实例对象访问到该成员
- **protected** 表示包级可见，但是其他包的子类也可以访问protected成员 但是不能访问private成员
- **default** 或者如果不加访问修饰符，默认表示只能包级可见
- **private** 表示只能在本类中使用，在其他类中不能调用

2.9 面向对象三大特性

- **封装 (Encapsulation)**
  - 将数据和基于数据的操作封装在一起，隐藏内部的细节，只保留一些对外的接口使其与外部发生联系。用户无需关心对象内部的细节，但可以通过对象对外提供的接口来访问该对象
  - 减少耦合: 可以独立地开发、测试、优化、使用、理解和修改
  - 减轻维护的负担: 可以更容易被程序员理解，并且在调试的时候可以不影响其他模块
  - 有效地调节性能: 可以通过剖析确定哪些模块影响了系统的性能
  - 提高软件的可重用性
  - 降低了构建大型系统的风险: 即使整个系统不可用，但是这些独立的模块却有可能是可用的
- **继承 (Inheritance)**
  - 继承实现了 **IS-A** 关系，例如 Cat 和 Animal 就是一种 IS-A 关系，因此 Cat 可以继承自 Animal，从而获得 Animal 非 private 的属性和方法
  - 继承应该遵循里氏替换原则，子类对象必须能够替换掉所有父类对象，子类拥有父类非 private 的属性和方法
  - 父类引用指向子类对象称为 **向上转型**
  - 在继承关系中，子类如果定义了一个与父类方法签名完全相同的方法，被称为重写（Override）
  - Override和Overload不同的是，如果方法名不同就是Overload，Overload方法是一个新方法；如果方法名相同，并且返回值也相同，就是`Override`
- **多态 (polymorphisn)**
  - 多态是指，针对某个类型的方法调用，其**真正执行的方法取决于运行时期实际类型的方法**
  - 多态分为编译时多态和运行时多态:
    - 编译时多态主要指方法的重载

        - 运行时多态指程序中定义的对象引用所指向的具体类型在运行期间才确定
    - 从字节码来说，方法的重载实际上重载的几个方法已经是相互独立的方法，方法通过方法名与参数列表区分；而重写的目的是为了实现子类与父类不同的表现，是多态。所以方法重载是静态的，是编译器行为，是在写代码期或者编译器可以确定执行哪个方法，而方法重写是动态的，是运行期行为，只有在执行期才知道是具体哪个类型来执行方法
    - Java实现多态有三个必要条件：继承、重写、向上转型
        - 继承：在多态中必须存在有继承关系的子类和父类。
        - 重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。
        - 向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该引用才能够具备调用父类的方法or调用子类的方法
- 作用
    - 应用程序不必为每一个派生类编写功能调用，只需要对抽象基类进行处理即可。大大提高程序的可复用性。//继承
    - 派生类的功能可以被基类的方法或引用变量所调用，这叫向后兼容，可以提高可扩充性和可维护性。 //多态的真正作用

2.10 重写和重载

- **重写(Override)**是子类对父类的允许访问的方法(不能用于静态方法和final、private)的实现过程进行重新编写
    - **方法名, 返回值和形参都不能改变**, **即外壳不变，核心重写**, 子类方法访问修饰符需要>=父类和接口方法访问修饰符
    - 重写的好处在于子类可以根据需要，定义特定于自己的行为。 也就是说子类能够根据需要实现父类的方法
    - **静态的方法可以被继承，但是不能重写。**如果父类中有一个静态的方法，子类也有一个与其方法名，参数类型，参数个数都一样的方法，并且也有static关键字修饰，那么该子类的方法会把原来继承过来的父类的方法隐藏，而不是重写，父类的方法和子类的方法是两个没有关系的方法，具体调用哪一个方法是看是哪个对象的引用；这种父子类方法也不在存在多态的性质。**只有普通的方法调用可以是多态的**
- **重载(Overload)**存在于同一个类中，指一个方法与已经存在的方法名称上相同，但是**参数类型、个数、顺序**至少有一个不同。应该注意的是，返回值不同，其它都相同不算是重载(会报错), 权限修饰符可以不同

2.11 Java 面向对象五大原则

- 遵循单一职责原则 SRP:
    - **a class should only have one responsibility**
    - 一个类只专注于做一件事；高内聚，低耦合
- 开闭原则 OCP：
    - **Open for Extension, Closed for Modification**
    - 如类，模块和函数应该对扩展开放(对提供方), 对修改关闭(对使用方)。用抽象构建框架，用实现扩展细节

- 里氏代换原则 LSP:
  - 客户端没有察觉情况下，子类必须能够替换它们的父类；子类可以扩展父类的功能，但不能改变父类原有的功能
  - 也就是说：子类继承父类时，除添加新的方法完成新增功能外，尽量不要重写父类的方法
- 接口分离原则 ISP：
  - **larger interfaces should be split into smaller ones**
  - 设计时采用多个与特定客户类有关的接口比采用一个通用的接口好, 大接口拆分为具体小接口，客户端只依赖于他们需要的接口
- 依赖倒置原则 DIP：
  - 高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。抽象不应该依赖于具体实现，具体实现应该依赖于抽象

2.12 变量生命周期

- 静态变量：存放在元空间, 生命从类加载开始，到类销毁
- 局部变量：局部变量必须初始化，在方法或代码块内有效，之外则无效，方法执行开始入栈时创建，执行完毕出栈时销毁
- 类成员变量：方法外部，类的内部定义的变量，存储在堆中的对象里面，由垃圾回收器负责回收, 生命周期从对象创建开始到对象销毁结束
- 字符串: 其对象的引用都是存储在栈中的，如果是编译期已经创建好(直接用双引号定义的)的就存储在常量池中; 如果是运行期（new出来的）才能确定的就存储在堆中

  new一个string，要先在常量池找object，没有的话先在常量池创建一个，然后再到堆中再创建一个拷贝对象

2.13 对象的生命周期

- **创建阶段**（参见new对象过程）
- **应用阶段**：至少被一个强引用持有着
- **不可见阶段**(非必须经历的阶段)：即使有强引用持有对象，但是这些强引用对于程序来说是不能访问的。例如在try{}里new了一个变量，try之后的语句就不可见
- **不可达阶段**:是指该对象不再被任何由gc root的强引用的引用链可达的状态。
- **收集阶段**: 当垃圾回收器发现该对象已经处于“不可达阶段”并且垃圾回收器已经对该对象的内存空间重新分配做好准备时，对象进入“收集阶段”。如果该对象已经重写了finalize()方法，并且没有被执行过，则执行该方法的操作。否则直接进入终结阶段。
- **终结阶段**:当对象执行完finalize()方法后仍然处于不可达状态时，该对象进入终结阶段。在该阶段，等待垃圾回收器回收该对象空间。
- **重新分配阶段**: 如果在完成上述所有工作完成后对象仍不可达，则垃圾回收器对该对象的所占用的内存空间进行回收或者再分配，该对象彻底消失。

2.14 JavaBean规范

- **JavaBean**是一种符合命名规范的 class ，它通过 getter 和 setter 来定义属性
- javaBean类必须是一个公共类，将其访问属性设置为public
- Javabean类不应有公共属性，属性都应该是 private
- 必须要有一个公共无参构造（写了带参构造必须添加一个，都不写则有默认）
- 为私有（private声明）属性提供符合命名规范的get/set方法
- 应该要实现serializeable 接口

2.15 抽象类与接口

- 接口 interface
  - 接口提供的都是方法，代表的是具备某方面的能力
  - 接口是抽象类的延伸，在 Java 8 之前，它可以看成是一个完全抽象的类，也就是说它不能有任何的方法实现。从 Java 8 开始，接口也可以拥有**默认的方法实现**，这是因为不支持默认方法的接口的维护成本太高了。在 Java 8 之前，如果一个接口想要添加新的方法，那么要修改所有实现了该接口的类
  - 接口的成员只能是 public 的，并且不允许定义为 private 或者 protected，接口的字段只能是 static 和 final 的
  - 接口的实现类和抽象类的子类只有**全部实现了接口或者抽象类中的方法**后才可以被实例化

```
public interface InterfaceExample {
    public int z = 0;       // Modifier 'public' is redundant for interface
fields
    // private int k = 0;   // Modifier 'private' not allowed here
    // protected int l = 0; // Modifier 'protected' not allowed here
    // private void fun3(); // Modifier 'private' not allowed here
    public default void func1();
}
public class InterfaceImplementExample implements InterfaceExample {
    @Override
    public void func1() {
        System.out.println("func1");
    }
}
// InterfaceExample ie1 = new InterfaceExample(); // 'InterfaceExample' is
abstract; cannot be instantiated
InterfaceExample ie2 = new InterfaceImplementExample();
ie2.func1();
```

  - 接口的实现类只能用接口定义的方法和变量，因为接口是封装好的
- 抽象 abstract
  - 抽象类和抽象方法都使用 abstract 关键字进行声明
  - 接口中用的更多的不是默认default方法，而是抽象方法

- 抽象类中的抽象方法**只是声明，不包含方法体**; 抽象类的子类**必须给出抽象类中的抽象方法的具体实现**，除非该子类也是抽象类
- 如果一个类中包含抽象方法，那么这个类必须声明为抽象类。**但抽象类不一定要有抽象方法**
- 抽象类和普通类最大的区别是，**抽象类不能被实例化 (会报错)**，**只有抽象类的非抽象子类才能实例化其子类**

```
public abstract class AbstractClassExample {
    protected int x;
    private int y;
    public abstract void func1();
    }
}
public class AbstractExtendClassExample extends AbstractClassExample {
    @Override
    public void func1() {
        System.out.println("func1");
    }
}
// AbstractClassExample ac1 = new AbstractClassExample();
// 'AbstractClassExample' is abstract; cannot be instantiated
AbstractClassExample ac2 = new AbstractExtendClassExample();
ac2.func1();
```

2.16 变量与方法

- 在调用子类构造方法之前会先调用父类没有参数的构造方法，其目的是?
  - 帮助子类做初始化工作。
- 一个类的构造方法的作用是什么？若一个类没有声明构造方法， 改程序能正确执行吗？为什么?
  - 主要作用是完成对类对象的初始化工作。可以执行。因为一个类即使没有声明构 造方法也会有默认的不带参数的构造方法。
- 构造方法有哪些特性?
  - 名字与类名相同；没有返回值，但不能用void声明构造函数；生成类的对象时自动执行，无需调用。
- 静态方法和实例方法有何不同?
  - 静态方法和实例方法的区别主要体现在两个方面:
  - 在外部调用静态方法时，可以使用"类名.方法名"的方式，也可以使 用"对象名.方法名"的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。
  - 静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此 限制
- 在一个静态方法内调用一个非静态成员为什么是非法的?
  - 由于静态方法可以不通过对象进行调用，因此在静态方法里，不能调用其他非静 态变量，也不可以访问非静态变量成员。
- 什么是方法的返回值？返回值的作用是什么?

- 方法的返回值是指我们获取到的某个方法体中的代码执行后产生的结果 (前提是该方法可能产生结果).
- 返回值的作用:接收出结果，使得它可以用于其他的操作!

2.17 Object类的方法

- equals()
  - 用于确认两个对象内容是否相同
  - 自反性、对称性、传递性、一致性、与 null 的比较都是false
  - 对于基本类型，== 判断两个值是否相等，基本类型没有 equals() 方法。
  - 对于引用类型，== 判断两个变量是否引用同一个对象，而 equals() 判断引用的对象是否等价
  - ```
    public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;

            EqualExample that = (EqualExample) o;

            if (x != that.x) return false;
            if (y != that.y) return false;
            return z == that.z;
        }
    ```
- hashCode()
  - hashCode() 返回散列值，而 equals() 是用来判断两个对象是否等价。等价的两个对象散列值一定相同，但是散列值相同的两个对象不一定等价
  - 在覆盖 equals() 方法时应当总是覆盖 hashCode() 方法，保证等价的两个对象散列值也相等
  - 如果不重写，那么所有collections都不能用
- toString()
  - 默认返回格式：对象的 class 名称 + @ + hashCode 的十六进制字符串
- clone()
  - 用于创建并返回一个对象的拷贝
  - clone() 是 Object 的 protected 方法，它不是 public，一个类不显式去重写 clone()，其它类就不能直接去调用该类实例的 clone() 方法
  - clone 方法是浅拷贝，对象内属性引用的对象只会拷贝引用地址，而不会将引用的对象重新分配内存。Object 本身没有实现 Cloneable 接口
    ```

```
public class CloneExample {
    private int a;
    private int b;

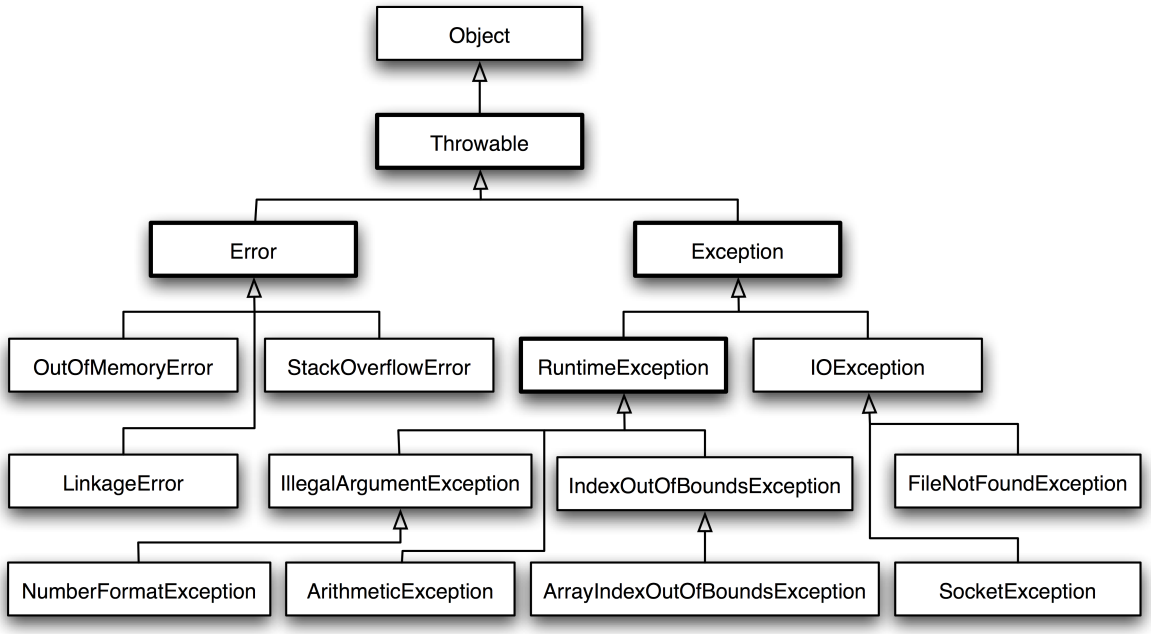
    @Override
    protected CloneExample clone() throws CloneNotSupportedException {
        return (CloneExample)super.clone();
    }
}
```

  - **浅拷贝** - 拷贝对象和原始对象的引用类型引用同一个对象
  - **深拷贝** - 会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象

- **GetClass**()：返回一个Class对象
- **wait(),wait(long),wait(long,int),notify(),notifyAll()** 在使用的时候要求在synchronize语句中使用
  - 每个java对象都有一把锁,wait/notify是对这个obj进行调用
  - Object lock = new Object(); lock.wait(); lock.notify()
  - wait()用于让当前线程失去操作权限，当前线程进入等待序列
  - notify()用于随机通知一个持有对象的锁的线程获取操作权限
  - notifyAll()用于通知所有持有对象的锁的线程获取操作权限
  - wait(long) 和wait(long,int)用于设定下一次获取锁的距离当前释放锁的时间间隔

2.18 Java 异常处理机制

- 什么是异常
  - 如果某个方法不能按照正常的途径完成任务，就可以通过另一种路径退出方法，在这种情况下会抛出一个封装了错误信息的对象
  - 这个方法会立刻退出同时不返回任何值，调用这个方法的其他代码也无法继续执行，异常处理机制会将代码执行交给异常处理器
  - 使用try、catch、finaly捕获异常，finaly中的代码一定会执行，捕获异常后程序会继续执行
  - 使用throws声明该方法可能会抛出的异常类型，出现异常后，程序终止
- **Throwable 是 所有错误或异常的超类**
  - Throwable有两个重要的子类：Error 和 Exception，各自都包含大量子类
  - 异常和错误的区别：异常能被程序本身可以处理，错误是无法处理



- **Error(错误)**
  - Error 类是指java 运行时系统的内部错误和资源耗尽错误，表示代码运行时 JVM出现的问题
  - 例如，Java虚拟机运行错误（Virtual MachineError）、（NoClassDefFoundError），OOM，StackOverFlow
  - 这些错误是不可查的，因为它们在应用程序的控制和处理能力之外
- **Exception(异常)**
  - Exception 类是程序本身可以处理的异常，有两个分支，运行时异常 RuntimeException 和 检查异常 CheckedException
  - 检查异常 CheckedException 一般是外部错误，这种异常都发生在编译阶段，Java 编译器会强制程序去捕获此类异常
    - 例如 IOException、SQLException、ClassNotFoundException
    - 当程序中可能出现这类异常，要么用try-catch语句捕获它，要么用throws子句声明抛出它，否则编译不会通过
  - 运行时异常 RuntimeException / UnCheckedException 表示"JVM 常用操作"引发的错误，不能被捕获
    - NullPointerException、ArithmeticException 和 ArrayIndexOutOfBoundException
- Throw 和 throws 的区别
  - 位置不同
    - throws 用在函数上，后面跟的是异常类，可以跟多个；而 throw 用在函数内，后面跟的是异常对象
  - 功能不同
    - throws 用来声明异常，让调用者只知道该功能可能出现的问题，可以给出预先的处理方式；throw 抛出具体的问题对象，执行到 throw，功能就已经结束了，跳转到调用者，并将具体的问题对象抛给调用者。也就是说 throw 语句独立存在时，下面不要定义其他语句，因为执行不到

        - throws 表示出现异常的一种可能性，并不一定会发生这些异常；throw 则是抛出了异常，执行 throw 则一定抛出了某种异常对象
        - 两者都是消极处理异常的方式，只是抛出或者可能抛出异常，但是不会由函数去处理异常，真正的处理异常由函数的上层调用处理
- Java 基础 - 异常机制详解

2.19 Java 反射

- **动态语言**是指程序在运行时可以改变其结构
    - 新的函数可以引进，已有的函数可以被删除等结构上的变化
    - 常见的 JavaScript 就是动态语言，除此之外 Ruby,Python 等也属于动态语言，而 C、C++则不属于动态语言
    - 从反射角度说 Java 属于半动态语言
- 反射机制概念
    - Java 中的反射机制是指在运行状态中，对于任意一个类都能够**知道这个类所有的属性和方法**；并且对于任意一个对象，都能够**调用它的任意一个方法和属性**；
    - 这种动态获取信息以及动态调用对象方法的功能成为 Java 语言的反射机制
- 编译时类型和运行时类型
    - 对象在运行是都会出现两种类型：编译时类型和运行时类型
    - 编译时的类型由声明对象时用的类型来决定，运行时的类型由实际赋值给对象的类型决定

```
// 编译时类型为 Person，运行时类型为 Student
Person p = new Student();
```

    - 编译时类型无法获取具体方法，此时就必须使用到反射了
- 每个类都有一个 Class 对象，包含了与类有关的信息。当编译一个新类时，会产生一个同名的 .class 文件，该文件保存着 Class 对象
- 获取 Class 对象的 3 种方法
    - 调用对象的 getClass() 方法来获取

```
Person p1 = new Person();
Class c1 = p1.getClass();
```

    - 调用某个类的 class 属性来获取该类对应的 Class 对象

```
Class c2 = Person.class;
```

    - 使用 Class 类中的 forName()静态方法(最安全/性能最好)

```
Class c3 = Class.forName("com.ys.reflex.Person");
```

- 当我们获得了想要操作的类的 Class 对象后，可以通过 Class 类中的方法获取并查看该类中的方法和属性

```
//获取 Person 类的所有方法信息
Method[] method = c1.getDeclaredMethods();
//获取 Person 类的所有成员属性信息
Field[] field = c1.getDeclaredFields();
//获取 Person 类的所有构造方法信息
Constructor[] constructor = c1.getDeclaredConstructors();
```

- Class 类包含方法：
  - getName()：获得类的完整名字。
  - getFields()：获得类的public类型的属性。
  - getMethods()：获得类的public类型的方法。
  - getConstructors()：获得类的public类型的构造方法。
  - newInstance()：通过类的**不带参数的构造方法**创建这个类的一个对象。
- 反射机制允许程序在运行时取得任何一个已知名称的class的内部信息，包括包括其修饰符(public,static,final)，类的变量，方法，并可于运行时改变变量或**调用methods**，降低代码的耦合度；还有动态代理的实现
- Class 和 java.lang.reflect 一起对反射提供了支持，java.lang.reflect 类库主要包含了以下三个类:
  - **Field** : 可以使用 get() 和 set() 方法读取和修改 Field 对象关联的字段
  - **Method** : 可以使用 invoke() 方法调用与 Method 对象关联的方法
  - **Constructor** : 可以用 Constructor 创建新的对象
- **反射的优点**
  - **可扩展性特性** ： 应用程序可以通过使用其完全限定名称创建可扩展性对象的实例来使用外部的、用户定义的类。
  - **类浏览器和可视化开发环境** ： 类浏览器需要能够枚举类的成员。可视化开发环境可以受益于利用反射中可用的类型信息来帮助开发人员编写正确的代码。
  - **调试器和测试工具** ： 调试器需要能够检查类的私有成员。测试工具可以利用反射系统地调用定义在类上的可发现集 API，以确保测试套件中的高水平代码覆盖率。
- **反射的缺点**
  - **性能开销** ： 由于反射涉及动态解析的类型，因此无法执行某些 Java 虚拟机优化。因此，反射操作的性能比它们的非反射对应物慢，并且应该避免在性能敏感的应用程序中经常调用的代码部分中。
  - **安全限制** ： 反射需要运行时权限，在安全管理器下运行时可能不存在。对于必须在受限安全上下文中运行的代码（例如在 Applet 中），这是一个重要的考虑因素。
  - **内部暴露** ： 由于反射允许代码执行在非反射代码中非法的操作，例如访问私有字段和方法，使用反射可能会导致意想不到的副作用，这可能会导致代码功能失调并可能破坏可移植性. 反射代码打破了抽象，因此可能会随着平台的升级而改变行为。

2.20 创建对象的五种方式

- 使用new关键字 -- 调用了构造函数
  - 在堆区分配对象需要的内存(空闲链表、指针碰撞):分配的内存包括本类和父类的所有实例变量，但不包括任何静态变量(因为它已在类加载过程初始好)
  - 对所有实例变量赋默认值：将方法区内对实例变量的定义拷贝一份到堆区，然后赋**默认值**
  - 设置对象头信息，包括gc分代年龄、对象的哈希值、锁信息
  - 执行实例初始化代码，赋**构造函数**中的值

    父类的静态代码块、父类的静态属性——>子类的静态代码块、子类的静态属性——>父类的初始化代码块、父类的实例属性——>父类的构造方法——>子类的初始化代码块、子类的实例属性——>子类的构造方法。（用"、"号分隔的优先级相同，执行顺序看代码顺序）
  - 如果有类似于Child c = new Child()形式的c引用的话，在**栈区**定义Child类型引用变量c，然后将**堆区对象的地址赋值**给它
  - 在创建子类对象的时候会调用父类的构造函数
- 使用Class类的newInstance方法 -- 调用了构造函数(无参构造)

```
// 内部调用Constructor的newInstance
Class testClass = Class.forName(“test.class”);
Object o = testClass.newInstance();
```

- 使用Constructor类的newInstance方法 -- 调用了构造函数(有参构造)
  - 通过这种方法可以选定构造方法创建实例
- 使用clone方法 -- 没有调用构造函数
- 使用反序列化 -- 没有调用构造函数

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.obj"));
Employee emp5 = (Employee) in.readObject();
```

2.21 Java注解

- 注解概念定义
  - Annotation (注解) 是 Java 提供的一种对元程序中元素关联信息和元数据 (metadata) 的途径和方法
  - Annotation 是一个接口，程序可以通过反射来获取指定程序中元素的 Annotation对象，然后通过该 Annotation 对象来获取注解中的元数据信息
  - 基本原则：注解不能直接干扰程序代码的运行，无论增加域删除注解，代码都能够正常运行
- 注解分类
  - 标注注解(marker annotation):没有元素的注解
  - 单值注解
  - 完整注解
- 4 种标准元注解

    - 元注解 (meta-annotation) 的作用是**负责注解其他注解**，它们被用来提供对其它 annotation 类型作说明
    - **@Target** 修饰的对象范围 (Where)
        - @Target说明了Annotation所修饰的对象范围
        - Annotation可被用于 TYPE(类、接口、枚举、Annotation 类型)、FIELD、Method、PARAMETER、CONSTRUCTOR(方法、构造方法、成员变量、枚举值)、LOCAL_VARIABLE(循环变量、catch 参数)
        - 在 Annotation 类型的声明中使用了 target 可更加明晰其修饰的目标
    - **@Retention** 定义被保留的时间长短 (When)
        - @Retention 表示需要在什么级别check注解信息，用于描述注解的生命周期，即被描述的注解在什么范围内有效
        - 取值（RetentionPoicy）有：在 `SOURCE` / `CLASS` / `RUNTIME` 中有效
    - **@Documented** 描述-javadoc
        - @Documented用于描述其它类型的 annotation 应该被作为被标注的程序成员的公共 API，因此可以被例如 javadoc 此类的工具文档化
    - **@Inherited** 阐述了某个被标注的类型是被继承的
        - @Inherited 元注解是一个标记注解，@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
- 标准注解
    - @Override:
        - 作用:保证编译时候Override函数的声明正确性
    - @Deprecated
        - 作用:对不应该在使用的方法添加注释，当编程人员使用这些方法时，将会在编译时显示提示信息与javadoc里的@deprecated标记有相同的功能，准确的说，它还不如javadoc @deprecated，因为它不支持参数
    - @SuppressWarnings
        - 作用:关闭特定的警告信息

2.22 Java Generic

- 概念定义
    - **泛型提供了编译时类型安全检测机制**，该机制允许程序员在编译时检测到非法的类型
    - 泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数，在实例化时作为参数指明类型
    - 类型通配符一般是使用 `?` 代替具体的类型参数，
- 如果不定义泛型类型时，泛型类型实际上就是 `Object`

```
// without generic
List names = new ArrayList();
names.add("Derek");
String name = (String) names.get(0);

// with generic
List<String> names = new ArrayList();
names.add("Derek");
String name = names.get(0);
```

- 三种使用方式，分别为：泛型方法、泛型类、泛型接口
  - 泛型方法 < E >
    - 泛型方法在调用时可以接收不同类型的参数，根据传递给泛型方法的参数类型，编译器适当地处理每一个方法调用
    - <? extends T>表示该通配符所代表的类型是 T 类型的子类
    - <? super T>表示该通配符所代表的类型是 T 类型的父类
  - 泛型类
    - 泛型类的声明和非泛型类的声明类似，除了在类名后面添加了类型参数声明部分
    - 和泛型方法一样，泛型类的类型参数声明部分也包含一个或多个类型参数，参数间用逗号隔开
    - 一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符

```
//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型。在实例化泛型类时，必须指定T的具体类型
public class Generic<T>{
    //key这个成员变量的类型为T,T的类型由外部指定
    private T key;
    public Generic(T key) {
        this.key = key;
    }
    public T getKey(){
        return key;
    }
}
//泛型的类型参数只能是类类型（包括自定义类），不能是简单类型
Generic<Integer> genericInteger = new Generic<Integer>(123456);
Generic<String> genericString = new Generic<String>("key_vlaue");
```

- Java在编译期间，所有的泛型信息都会被擦掉.在泛型类被**类型擦除**的时候，之前泛型类中的类型参数部分如果没有指定上限，如 则会被转译成普通的 Object 类型，如果**指定了上限**如`<T extends HashMap>`则类型参数就被替换成类型上限
- 类型擦除
  - Java 中的泛型基本上都是在编译器这个层次来实现的，在生成的 Java 字节代码中是不包含泛型中的类型信息的

    - 使用泛型的时候加上的类型参数，会被编译器在编译的时候去掉，这个过程就称为类型擦除 (Type Erasure)
        - 如在代码中定义的 `List<Object>` 和 `List<String>` 等类型，在编译之后都会变成 List
        - JVM 看到的只是 List，而由泛型附加的类型信息对 JVM 来说是不可见的
    - Java使用类型擦除实现泛型，导致：
        - 编译器把类型 `<T>` 视为 `Object`
        - 编译器根据 `<T>` 实现安全的强制转型 Type Cast
    - Java泛型的局限
        - `<T>` 不能是基本类型
        - 无法取得带泛型的 `Class`
        - 无法判断带泛型的类型
- Java 基础 - 泛型机制详解
- 廖雪峰 - 什么是泛型

2.23 Java序列化

- 保存(持久化)对象及其状态到内存或者磁盘，**二进制流**
    - Java 平台允许我们在内存中创建可复用的 Java 对象，但一般情况下，只有当 JVM 处于运行时，这些对象才可能存在，即，这些对象的生命周期不会比 JVM 的生命周期更长
    - 在现实应用中，就可能要求在JVM停止运行之后能够保存(持久化)指定的对象，并在将来重新读取被保存的对象
    - Java 对象序列化就能够帮助我们实现该功能
- 序列化对象保持字节数组 - 不保存静态成员
    - 使用 Java 对象序列化，在保存对象时，会把其状态保存为一组字节，在未来，再将这些字节组装成对象
    - 注意地是，对象序列化保存的是对象的”状态”，即它的成员变量。即对象序列化不会关注类中的静态变量
- 序列化用户远程对象传输
    - 除了在持久化对象时会用到对象序列化之外，当使用 RMI(远程方法调用)，或在网络中传递对象时， 都会用到对象序列化
    - Java序列化API为处理对象序列化提供了一个标准机制，该API简单易用
- Serializable 实现序列化
    - Java 中，只要一个类实现了 java.io.Serializable 接口，那么它就可以被序列化
    - ObjectOutputStream 和 ObjectInputStream 对对象进行序列化及反序列化
    - writeObject 和 readObject 自定义序列化策略
    - Serializable interface is used to perform serialization.

```
// class object e implements Serializable
FileOutputStream fileOut = new FileOutputStream("./file.ser");
ObjectOutputStream out = new ObjectOutputStream(fileOut);
out.writeObject(e);
out.close();
fileOut.close();
```

- The content can be restored using **deserialization**.

```
// class object e implements Serializable
FileInputStream fileIn = new FileInputStream(("./file.ser");
ObjectInputStream in = new ObjectInputStream(fileIn);
e = (e转型) in.readObject();
in.close();
fileIn.close();
```

2.24 IO

- Stream
  - based on bit units: Bytestream, CharactenStream (16 bits)
  - based on direction: InputStream, OutputStream
  - ReaderWritep
- Abstract Level ByteStream CharacterStream
- Input InputStream, Reader
- Output OutputStream, Writer
- There are overall more than 40 types of I0 streams in Java.
  - abstract super cLass, ButaStream(in), Butestream(out), Characterstream(in), Characterstreom(out)
  - file access,
  - array access,
  - pipe access,
  - string access,
  - buffered stream,
  - transfer stream,
  - object stream,
  - print stream,
  - push back input,
  - Special Stream,

- InputStreomFileInputstreamByteArpayInputStreomPipedInputStreamn/a
  BufferedInputStream
  n/a
  ObjectInputStreomFilterInputstreomn/a
  PushbackInputStreomDataInputStream
- OutputStreamFileOutputSteeamByteArrayOutputStreamPipedOutputStreamn/a
  BufferedOutputStreamn/a
  ObjectOutputstreomFilterQutputstreamPrintStreemn/a
  DataQutputstream
- Reader
  FileReaderCh蚱报臆鲨货惫殆
  ArrayReaderPipedReaderStringReaderBufferedReaderInputStreamReadern/a
  FilterReadern/aPushbackReadern/a
  Characterstreom(in)RriterFileWriterCharArrayWriterPipedWriterStringWriterBufferedWriterOutputstneamwmiternla
  FilterWriterPrintWritern/an/a

3. ♥ Java 集合框架 ♥

3.1 Java Collections Framework

- **Java Collections Framework(JCF)** 为Java开发者提供了通用的容器
- **Java Collection Framework** is a combination of classes and interface, which is used to store and manipulate the data in the form of objects. It provides various classes such as ArrayList, Vector, Stack, and HashSet, etc. and interfaces such as List, Queue, Set, etc. for this purpose
- Java容器里只能放对象
  - 对于基本类型(int, long, float, double等)，需要将其包装成对象类型后(Integer, Long, Float, Double等)才能放到容器里
- 集合框架主要包括 Collection 和 Map，Collection 存储着对象的集合，而 Map 存储着键值对(两个对象)的映射表
- All types of collection 实现了Iterable，并声明了foreach方法

3.2 JCF Various Interfaces

- Collection interface (java.util.Collection) and Map interface (java.util.Map) are the interfaces of Java Collections Framework.
- **Collection interface**:
  - Collection (java.util.Collection) is the primary interface, and every collection must implement this interface.

```
public interface Collection<E> extends Iterable
```

- **List interface**:
  - List interface extends the Collection interface, and it is an ordered collection of objects.
  - It contains duplicate elements. It also allows random access of elements.

```
public interface List<E> extends Collection<E>
```

- **Set interface:**
  - Set (java.util.Set) interface is a collection which cannot contain duplicate elements.
  - It can only include inherited methods of Collection interface

```
public interface Set<E> extends Collection<E>
```

- **Queue interface:**
  - Queue (java.util.Queue) interface defines queue data structure, which stores the elements in the form FIFO.

```
public interface Queue<E> extends Collection<E>
```

- **Dequeue interface:**
  - It is a double-ended-queue. It allows the insertion and removal of elements from both ends.
  - It implants the properties of both Stack and queue so it can perform LIFO (Last in first out) stack and FIFO (first in first out) queue, operations.

```
public interface Dequeue<E> extends Queue<E>
```

- **Map interface:**
  - A Map (java.util.Map) represents a key, value pair storage of elements.
  - Map interface does not implement the Collection interface. It can only contain a unique key but can have duplicate elements. There are two interfaces which implement Map in java that are Map interface and Sorted Map.

```
public interface Map<K,V>
```

3.3 List - ArrayList

- ArrayList实现了List接口，是顺序容器，即元素存放的数据与放进去的顺序相同，允许放入null元素，底层通过数组实现
- 除该类未实现同步外，其余跟Vector大致相同，Vector和 ArrayList 类似，但它是线程安全的，大多方法用synchronize修饰
- 每个ArrayList都有一个容量(capacity)，表示底层数组的实际大小，容器内存储元素的个数不能多于当前容量
- 当向容器中添加元素时，若ArrayList已有的存储能力满足最低存储要求，则返回add直接添加元素

- 如果容量不足，最低要求的存储能力>ArrayList已有的存储能力，容器会**扩容**，需要调用 grow()方法进行扩容
    - 当ArrayList扩容的时候，首先会设置新的存储能力为原来的**1.5倍**，如果扩容之后仍小于必要存储要求minCapacity，则取值为minCapacity
    - 若新的存储能力大于MAX_ARRAY_SIZE，则取值为Integer.MAX_VALUE
    - 确定ArrayList扩容后的新存储能力后，调用Arrays.copyof() 方法进行对原数组的复制，底层通过调用System.arraycopy() 方法（native修饰）进行复制，达到扩容的目的
- size(), isEmpty(), get(), set()方法均能在常数时间内完成，add()方法的时间开销跟插入位置有关，addAll()方法的时间开销跟添加元素的个数成正比
- 基于动态数组实现，支持O(1)随机访问。但**插入删除需要搬动数据**
    - **删除元素**需要调用 System.arraycopy() 将 index+1 后面的元素都复制到 index 位置上，该操作的时间复杂度为 O(N)，ArrayList 删除元素的代价是非常高的
- 线程安全的arrayList
    - 可以使用 Collections.synchronizedList(); 得到一个**线程安全**的 ArrayList。大多方法synchronize竞争同一把锁

```
List<String> list = new ArrayList<>();
List<String> synList = Collections.synchronizedList(list)
```

- CopyOnWriteArrayList 读写分离
    - 往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素
    - 所以CopyOnWrite容器也是一种读写分离的思想，**大大提高了读操作的性能，因此很适合读多写少的应用场景**
    - 增删改时，需要加锁，否则copy出很多个副本
    - 但是 CopyOnWriteArrayList 有其缺陷：
        - 内存占用：在写操作时需要复制一个新的数组，使得内存占用为原来的两倍左右
        - 数据不一致：读操作不能读取实时性的数据，因为部分写操作的数据还未同步到读数组中
        - 所以 CopyOnWriteArrayList 不适合内存敏感以及对实时性要求很高的场景

3.4 List - LinkedList

- LinkedList同时实现了List接口和Deque接口，既可以看作一个顺序容器，又可以看作一个双端队列(Queue)，底层通过链表实现
- 只能遍历顺序访问，但是在链表中间插入和删除元素比ArrayList略方便一点
- 当需要使用栈或者队列时，可以考虑使用LinkedList，一方面是因为Java官方已经声明不建议使用Stack类，更遗憾的是，Java里根本没有一个叫做Queue的类
- 关于栈或队列，现在的首选是ArrayDeque，它有着比LinkedList (当作栈或队列使用时) 有着更好的性能

3.5 List - Vector

- Vector 与 ArrayList 一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写 Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此访问它比访问 ArrayList 慢

3.6 Set - HashMap & HashSet

- **Hash**
  - JDK 1.8 中，是通过 hashCode() 的高 16 位异或低 16 位实现的，主要是从速度，功效和质量来考虑的，减少系统的开销，也不会造成因为高位没有参与下标的计算，保留高16位与低16位的特性，增大散列程度，减少哈希碰撞
  - hash函数计算出的值，通过indexFor与运算（return h & (length-1);）来获取实际的存储位置
- **Map**
  - Key -> Value 组成 entry, Map.of()可以创建不可更改的Map
- **HashMap**
  - `HashMap` 实现了Map接口
    - 即允许放入 `key` 为 `null` 的元素，也允许插入 `value` 为 `null` 的元素
    - 但最多只允许一条记录的 `key` 为 `null`，允许多条记录的 `value` 为 `null`
    - 线程不安全，除该类未实现同步外，其余跟 `Hashtable` 大致相同
    - 如果需要满足线程安全，可以用 Collections.synchronizedMap，或者使用 ConcurrentHashMap
  - `HashMap` 根据键的 hashCode 值存储数据，可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的
  - 跟 `TreeMap` 不同，`HashMap` 不保证元素顺序，根据需要 `HashMap` 可能会对元素重新哈希，元素的顺序也会被重新打散，因此不同时间迭代同一个 `HashMap` 的顺序可能会不同
  - Java7 HashMap 结构
    - HashMap 里面是一个数组，然后数组中每个元素是一个单向链表
    - 链表上每个元素是一个是嵌套类 Entry 的实例，Entry 包含四个属性：key, value, hash 值和用于单向链表的 next
    - 查找的时候，根据 hash 值我们能够快速定位到数组的具体下标，但是之后需要顺着链表一个个比较下去才能找到需要的，时间复杂度取决于链表的长度，为O(n)
  - Java8 HashMap 结构
    - 最大的不同就是利用了红黑树，所以其由数组+链表+红黑树组成
    - 当链表中的元素超过了 8 个以后，会将链表转换为红黑树，在这些位置进行查找的时候可以降低时间复杂度为 O(logN)
  - HashMap的扩容机制
  - HashMap存取原理：
    - 计算key的hash值，然后进行二次hash，根据二次hash结果找到对应的索引位置

        - 如果这个位置有值，先进性equals比较，若结果为true则取代该元素，若结果为false，就使用高低位平移法将节点插入链表
    - 根据对冲突的处理方式不同，哈希表有两种实现方式，一种开放地址方式(Open addressing)，另一种是冲突链表方式(Separate chaining with linked lists)
- **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 存储元素的顺序并不是按照存入时的顺序，而是按照哈希值来存的所以取数据也是按照哈希值取得
    - 元素的哈希值是通过元素的hashcode 方法来获取，HashSet 首先判断两个元素的哈希值
        - 如果哈希值一样，接着会比较equals 方法
        - 如果 equals 为 true，HashSet 就视为同一个元素
        - 如果 equals 为 false 就不是同一个元素

3.7 Set - ConcurrentHashMap

- Segment
    - ConcurrentHashMap 和 HashMap 思路是差不多，但是因为它支持并发操作，所以要更复杂
    - 整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表”部分“或”一段“，很多地方都会将其描述为分段锁
- 线程安全（Segment 继承 ReentrantLock 加锁）
    - ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全
- Java7 ConcurrentHashMap
    - concurrencyLevel：并行级别、并发数、Segment 数，默认是 16，ConcurrentHashMap 有 16 个 Segments
    - 理论上最多可以同时支 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上
    - 这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的
    - 具体到每个 Segment 内部，每个 Segment 很像之前HashMap，是数组加单向链表
- Java8 ConcurrentHashMap
    - Java8 对 ConcurrentHashMap 实现已经摒弃了Segment的概念，而是直接用Node数组+链表+红黑树的数据结构来实现，并发控制使用Synchronized和CAS来操作，整个看起来就像是优化过且线程安全的HashMap

3.8 Set - HashTable

- Hashtable 是legacy类，很多映射的常用功能与 HashMap 类似，不同的是它承自 Dictionary 类, 并且是线程安全的，任一时间只有一个线程能写 Hashtable，并发性不如 ConcurrentHashMap，因为 ConcurrentHashMap 引入了分段锁
- Hashtable 不建议在新代码中使用，不需要线程安全的场合可以用 HashMap 替换，需要线程安全的场合可以用 ConcurrentHashMap 替换

3.9 Set - TreeMap & TreeSet

- **TreeMap**
  - `TreeMap`实现了SortedMap接口，能够把它保存的记录根据键排序，底层通过红黑树(Red-Black tree)实现，根据元素的comparedTo()或者创建TreeSet 时提供的 Comparator 进行排序，默认是按键值的升序排序，也可以指定排序的比较器，当用 Iterator 遍历 `TreeMap` 时，得到的记录是排过序的
  - `containsKey()`，`get()`，`put()`，`remove()`都有着`log(n)`的时间复杂度
  - 支持有序性操作，例如根据一个范围查找元素的操作
  - 但是查找效率不如 HashSet，HashSet 查找的时间复杂度为 O(1)，TreeSet 则为 O(logN)
  - 红黑树左旋的过程是将`x`的右子树绕`x`逆时针旋转，使得`x`的右子树成为`x`的父亲，同时修改相关节点的引用

```
//Rotate Left
private void rotateLeft(Entry<K,V> p) {
    if (p != null) {
        Entry<K,V> r = p.right;
        p.right = r.left;
        if (r.left != null)
            r.left.parent = p;
        r.parent = p.parent;
        if (p.parent == null)
            root = r;
        else if (p.parent.left == p)
            p.parent.left = r;
        else
            p.parent.right = r;
        r.left = p;
        p.parent = r;
    }
}
```

  - 红黑树右旋的过程是将`x`的左子树绕`x`顺时针旋转，使得`x`的左子树成为`x`的父亲，同时修改相关节点的引用

```
//Rotate Right
private void rotateRight(Entry<K,V> p) {
    if (p != null) {
        Entry<K,V> l = p.left;
        p.left = l.right;
        if (l.right != null) l.right.parent = p;
        l.parent = p.parent;
        if (p.parent == null)
            root = l;
        else if (p.parent.right == p)
            p.parent.right = l;
        else p.parent.left = l;
        l.right = p;
```

```
            p.parent = l;
        }
    }
```

- **TreeSet**
  - `TreeSet`是对`TreeMap`的简单包装，对`TreeSet`的函数调用都会转换成合适的`TreeMap`方法
  - TreeSet是使用二叉树的原理对新 add()的对象按照指定的顺序排序（升序、降序），每增加一个对象都会进行排序，将对象插入的二叉树指定的位置
  - Integer 和 String 对象都可以进行默认的 TreeSet 排序，而自定义类的对象是不可以的，己定义的类必须实现 Comparable 接口，并且覆写相应的 compareTo()函数，才可以正常使用
  - 在覆写 compare()函数时，要返回相应的值才能使 TreeSet 按照一定的规则来排序
  - 比较此对象与指定对象的顺序。如果该对象小于、等于或大于指定对象，则分别返回负整数、零或正整数

3.10 Set - LinkedHashMap & LinkedHashSet

- **LinkedHashMap**
  - `LinkedHashMap`是`LinkedList`和`HashMap`的混合体，同时满足`LinkedList`和`HashMap`的某些特性
  - `LinkedHashMap`是`HashMap`的直接子类，二者唯一的区别是LinkedHashMap在HashMap的基础上，采用**双向链表**(doubly-linked list)的形式将所有entry连接起来，这样是为保证元素的迭代顺序跟**插入顺序相同**
  - `LinkedHashMap`跟`HashMap`完全一样，多了header指向双向链表的header，该双向链表的迭代顺序就是entry的插入顺序
  - 除了可以保存插入顺序，这种结构还有一个好处 : 迭代LinkedHashMap时不需要像HashMap那样遍历整个table，而只需要直接遍历header指向的双向链表即可，也就是说LinkedHashMap的迭代时间就只跟entry的个数相关，而跟table的大小无关

    ```
    LinkedHashMap<String,Integer> linkmap = new LinkedHashMap<>()
    ```

- **LinkedHashSet**
  - `LinkedHashSet`是对`LinkedHashMap`的简单包装，对`LinkedHashSet`的函数调用都会转换成合适的`LinkedHashMap`方法
  - `LinkedHashSet`是继承于 HashSet、又基于 LinkedHashMap 来实现的
    - LinkedHashSet 底层使用 LinkedHashMap 来保存所有元素，继承于 HashSet，其所有的方法操作上又与 HashSet 相同
    - 因此 LinkedHashSet 的实现上非常简单，只提供了四个构造方法，并通过传递一个标识参数，调用父类的构造器，底层构造一个 LinkedHashMap 来实现，在相关操作上与父类 HashSet 的操作相同，直接调用父类 HashSet 的方法即可

3.11 Queue - Deque & Array

- Java里有一个叫做Stack的类，却没有叫做Queue的类(它是个接口名字)。当需要使用栈时，Java已不推荐使用Stack，而是推荐使用更高效的ArrayDeque
- Queue接口继承自Collection接口，除了最基本的Collection的方法之外，它还支持额外的insertion, extraction和inspection操作
- Deque是"double ended queue", 表示双向的队列。Deque 继承自 Queue接口，除了支持Queue的方法之外，还支持insert, remove和examine操作，由于Deque是双向的，所以可以对队列的头和尾都进行操作
- 当把Deque当做FIFO的queue来使用时，元素是从deque的尾部添加，从头部进行删除的
- ArrayDeque底层通过数组实现，为了满足可以同时在数组两端插入或删除元素的需求，该数组还必须是循环的，即循环数组(circular array)，也就是说数组的任何一点都可能被看作起点或者终点。
- ArrayDeque是非线程安全的(not thread-safe)，当多个线程同时使用的时候，需要程序员手动同步；另外，该容器不允许放入null元素

3.12 Queue - PriorityQueue

- PriorityQueue，即优先队列的作用是能保证每次取出的元素都是队列中权值最小的
- 其通过堆实现，具体说是通过完全二叉树(complete binary tree)实现的小顶堆(任意一个非叶子节点的权值，都不大于其左右子节点的权值)，也就意味着可以通过数组来作为PriorityQueue的底层实现
- 父节点和子节点的编号是有联系的，更确切的说父子节点的编号之间有如下关系:
  - leftNo = parentNo*2+1 *
  - rightNo = parentNo2+2
  - parentNo = (nodeNo-1)/2
- offer()

```
public boolean offer(E e) {
    if (e == null)//不允许放入null元素
        throw new NullPointerException();
    modCount++;
    int i = size;
    if (i >= queue.length)
        grow(i + 1);//自动扩容
    size = i + 1;
    if (i == 0)//队列原来为空，这是插入的第一个元素
        queue[0] = e;
    else
        siftUp(i, e);//调整
    return true;
}
```

- siftUp()

```
//siftUp()
private void siftUp(int k, E x) {
    while (k > 0) {
        int parent = (k - 1) >>> 1;//parentNo = (nodeNo-1)/2
        Object e = queue[parent];
        if (comparator.compare(x, (E) e) >= 0)//调用比较器的比较方法
            break;
        queue[k] = e;
        k = parent;
    }
    queue[k] = x;
}
```

- PriorityQueue源码解析

3.12 总结

- Collection
  - List
    - ArrayList
      - 排序有序，可重复
      - 底层使用数组
      - 查询速度快，get()和set()快，增删慢
      - 线程不安全
      - 扩容是当前容量*1.5+1
    - LinkedList
      - 排序有序，可重复
      - 底层使用双向循环列表
      - 查询速度慢，增删快，add()和remove()快
      - 线程不安全
    - Vector
      - 排序有序，可重复
      - 底层使用数组
      - 查询速度快，增删慢
      - 线程安全，效率低
      - 扩容是当前容量*2
  - Set
    - HashSet
      - 排列无序，不可重复
      - 底层使用Hash表实现
      - 存取速度快

        - 内部是HashMap
    - TreeSet
        - 排列无序，不可重复
        - 底层使用二叉树实现
        - 排序存储
        - 内部是TreeMap的SortedSet
    - LinkedHashSet
        - 采用hash表存储，并用双向链表记录插入顺序
  - Queue
    - 两端出入的List，所以也可以用链表或者数组实现
- Map
  - HashMap
    - 键不可重复,值可重复
    - 底层哈希表
    - 线程不安全
    - 允许key值为null，value也可以为null
  - hashtable
    - 键不可重复值可重复
    - 底层哈希表
    - 线程安全
    - key、value都不允许为null
  - TreeMap
    - 键不可重复值可重复
    - 底层二叉树

4. ♥ Java 多线程 ♥

4.1 多线程概念

- 多线程是什么
  - Multithreading is a process of executing multiple threads simultaneously, used to obtain the multitasking.
  - It consumes less memory and gives the fast and efficient performance.
  - **Main advantages**: Threads share the same address space. The thread is lightweight. The cost of communication between the processes is low.
- JUC 是什么

- JUC是java.util.concurrent包的简称，在Java5.0添加，目的就是为了更好的支持高并发任务。让开发者进行多线程编程时减少竞争条件和死锁的问题

- Process 和 Thread 区别
  - **Process 进程**:
    - 一个运行中的程序的集合，进程拥有自己的资源空间，**一个进程包含多个线程**，进程拥有自己的资源空间
  - **Thread 线程**:
    - 系统调度的最小单元是线程 thread ，也叫轻量级进程 (lightweight process)
    - 每个线程都拥有各自的计数器、堆栈和局部变量等属性，并且能够访问共享的内存变量
  - 进程在内存中有不同的地址空间；线程包含共享地址空间
  - 线程的 context switching 比进程快，因为上下文转换时要保存和切换的东西少
  - 进程间通信 (inter-process communication) is **slower and expensive** than 线程间通信 (inter-thread communication)

- 进程间如何通信
  - 进程间通过管道、 命名管道(FIFO)、消息队列、信号量机制、共享内存
  - 管道(在内存中)：它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）
  - 命名管道(在内存中)：任何进程间都能通讯，但速度慢

    ```
    mkfifo(client_fifo_name, FILE_MODE)
    ```

  - 消息队列：消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级
    - 独立于发送与接收进程,从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取，而不像 FIFO 那样只能默认地接收

    ```
    int msgget(key_t key, int msgflg);
    ```

  - 信号量：是一个计数器，不能传递复杂消息，只能用来同步

    ```
    int semget (key_t key, int nsems, int flags)
    ```

  - 共享内存：速度快，缺点：要进行同步
    - 进程地址空间相互独立，每个进程各自有不同的用户地址空间。一个进程的全局变量在另一个进程中都看不到，所以进程之间空间不能相互访问，要交换数据必须通过内核，在内核(虚拟地址里的高地址1G空间)中开辟一块缓冲区。之前的三四种通信必须借助内核，写入数据时都需要从进程复制到内核，读取时从内核复制到进程
    - 将同一块内存区域映射到共享它的不同进程的地址空间中，使得这些进程间的通信就不需要再经过内核，只需对该共享的内存区域进程操作

- 线程间如何通信
  - 共享内存，volatile

    - lock，semaphore
    - 管道通信
        - 管道流只能在两个线程之间传递数据
        - 管道流只能实现单向发送，如果要两个线程之间互通讯，则需要两个管道流
- 上下文切换
    - 上下文切换 (进程切换或任务切换) 是指 CPU 从一个进程 (或线程) 切换到另一个进程 (或线程)
    - 上下文是指某一时间点 CPU 寄存器和程序计数器的内容
    - CPU通过为每个线程分配CPU时间片来实现多线程机制, 时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行。CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务
    - 切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态
        - 任务从保存到再加载的过程就是一次上下文切换
    - 上下文切换通常是计算密集型的，意味着此操作会消耗大量的 CPU 时间，故线程也不是越多越好
- 如何减少上下文切换
    - 减少上下文切换的方法有无锁并发编程、CAS算法、使用最少线程和使用协程
    - 无锁并发编程。多线程竞争锁时，会引起上下文切换，所以多线程处理数据时，可以用一些办法来避免使用锁，如将数据的ID按照Hash算法取模分段，不同的线程处理不同段的数据
    - CAS算法。Java的Atomic包使用CAS算法来更新数据，而不需要加锁
    - 使用最少线程。避免创建不需要的线程，比如任务很少，但是创建了很多线程来处理，这样会造成大量线程都处于等待状态
    - 协程：在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换
- 线程共享进程的那些资源
    - 堆（是在进程空间中开辟出来）
    - 全局变量, 静态变量
    - 进程代码段、进程的公有数据(利用这些共享的数据，线程可实现相互通讯)、进程打开的文件描述符
- 并发与并行的区别
    - 并发：多线程操作单个CPU交替执行
    - 并行：多个线程在多个CPU上同时进行, 使用线程池操作
- wait与sleep的区别
    - 来自不同的类： wait来自object类, sleep来自线程类
    - 关于锁的释放：wait会释放锁, sleep不会释放锁
    - 使用的范围不同： wait必须在同步代码块中， sleep可以在任何地方睡眠

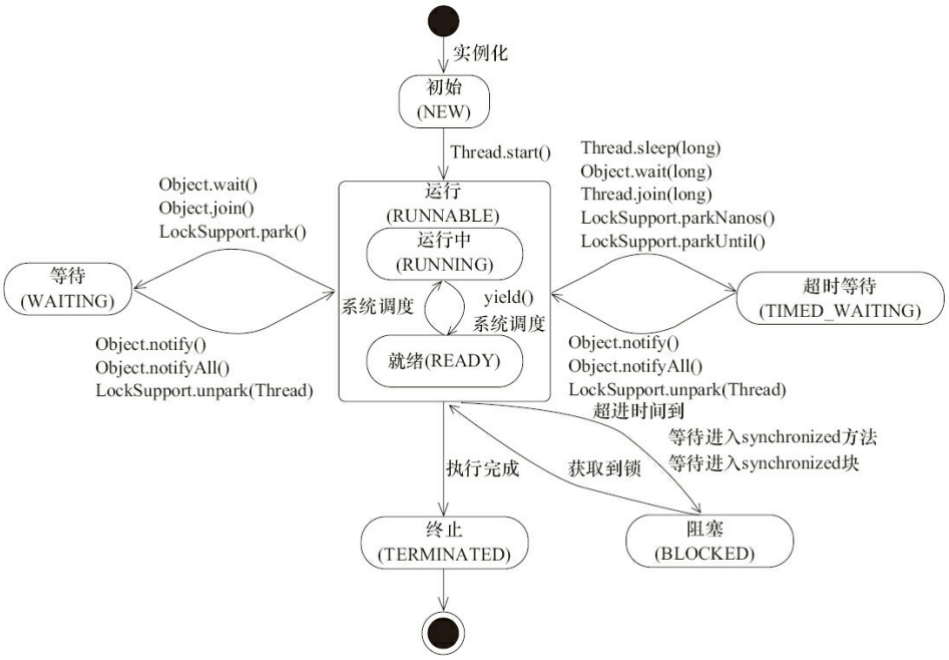
4.2 为什么需要多线程

- CPU、内存、I/O 设备的速度有极大差异，需要合理利用 CPU 的高性能，平衡这三者的速度差异
- 使用多线程的原因：
  - 大多数计算机都更加擅长并行计算，拥有更多的处理器核心
  - 缩短响应时间，提升用户体验
  - Java提供良好、考究并且一致的编程模型，开发人员能够更加专注于问题的解决，而不是多线程化
- 多线程导致的问题
  - CPU 增加了缓存，以均衡与内存的速度差异；// 导致 `可见性` 问题
  - 操作系统增加了进程、线程，以分时复用 CPU，进而均衡 CPU 与 I/O 设备的速度差异；// 导致 `原子性` 问题
  - 编译程序优化指令执行次序，使得缓存能够得到更加合理地利用。// 导致 `有序性` 问题

4.3 线程状态转换

- Java线程在运行的生命周期中可能处于6种不同的状态，给定的一个时刻只能处于其中的一个状态
  - **初始状态 (New)** 线程创建后尚未启动，还没有调用start() 方法没有执行资格
  - **运行状态 (Runnable)** 可能就绪正在等待 CPU 时间片，也可能正在运行
  - **阻塞状态 (Blocking)** 因某种原因放弃了 cpu 使用权，即让出了 cpu timeslice，暂时停止运行
    - 等待阻塞 (o.wait->等待待队列) 执行 o.wait()方法，JVM 会把该线程放入等待队列(waitting queue)
    - 同步阻塞 (lock->锁池) 获取对象同步锁时，若该同步锁被别的线程占用，JVM 会把该线程放入锁池(lock pool)
    - 其他阻塞 (sleep/join) 执行 Thread.sleep(long ms)或 t.join()方法，或者发出了 I/O 请求时，JVM 会把该线程置为阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入可运行(runnable)状态
  - **等待状态 (Waiting)** 等待其它线程显式地唤醒，否则不会被分配 CPU 时间片
    - 没有设置 Timeout 参数的 Object.wait() 方法, 等待Object.notify() / Object.notifyAll()
    - 没有设置 Timeout 参数的 Thread.join() 方法, 等被调用的线程执行完毕
  - **超时等待状态 (Timed Waiting)** 无需等待其它线程显式地唤醒，在一定时间之后会被系统自动唤醒
    - Thread.sleep() 方法时限内结束
    - 设置了 Timeout 参数的 Object.wait() 方法, 等待时间结束 / Object.notify() / Object.notifyAll()
    - 设置了 Timeout 参数的 Thread.join(), 等待方法时间结束 / 被调用的线程执行完毕
  - **终止状态 (Terminated)** 可以是线程结束任务之后自己结束，或者产生了异常而结束
    - 正常结束，run()或 call()方法执行完成，线程正常结束
    - 异常结束，线程抛出一个未捕获的 Exception 或 Error
    - 调用 stop，直接调用该线程的 stop()方法来结束该线程—该方法通常容易导致死锁

- 线程在自身的生命周期中，并不是固定地处于某个状态，而是随着代码的执行在不同的状态之间进行切换



- Daemon线程
  - Daemon (守护线程)是一种支持型线程，因为它主要被用作程序中后台调度以及支持性工作
  - 当所有非Daemon线程结束时，Java虚拟机将会退出，同时会杀死所有为Daemon线程
  - 可以通过调用 Thread.setDaemon(true) 方法将一个线程设置为Daemon线程
    - main() 属于非Daemon线程
    - Daemon属性需要在启动线程之前设置，不能在启动线程之后设置
  - 构建Daemon线程时，不能依靠finally块中的内容来确保执行关闭或清理资源的逻辑, finally块不一定会执行

4.4 线程实现方式

- 继承 Thread 类
  - Thread 类本质上是实现了 Runnable 接口的一个实例，代表一个线程的实例，需要实现 run() 方法
  - Thread 类的 start()实例方法是一个 native 方法，它将启动一个新线程，并执行 run()方法

```
public class MyThread extends Thread {
    public void run() {
        // ...
    }
}
public static void main(String[] args) {
    MyThread mt = new MyThread();
    mt.start();
}
```

- 实现 Runnable 接口
  - 如果已经 extends 另一个类，无法直接 extends Thread，可以实现Runnable 接口，需要实现 run() 方法
  - 通过 Thread 调用 start() 方法来启动线程

```
public class MyRunnable implements Runnable {
    public void run() {
        // ...
    }
}
public static void main(String[] args) {
    MyRunnable instance = new MyRunnable();
    Thread thread = new Thread(instance);
    thread.start();
}
```

- 实现 Callable 接口
  - 有返回值的任务必须实现 Callable 接口，类似的，无返回值的任务必须实现 Runnable 接口
  - 返回值通过 FutureTask 进行封装

```
public class MyCallable implements Callable<Integer> {
    public Integer call() {
        return 123;
    }
}
public static void main(String[] args) throws ExecutionException,
InterruptedException {
    MyCallable mc = new MyCallable();
    FutureTask<Integer> ft = new FutureTask<>(mc);
    Thread thread = new Thread(ft);
    thread.start();
    System.out.println(ft.get());
}
```

- 继承 Thread VS 实现Runnable/Callable 接口
  - 实现接口会更好一些，因为Java 不支持多重继承，因此继承了 Thread 类就无法继承其它类，但是可以实现多个接口；
  - 类可能只要求可执行就行，继承整个 Thread 类开销过大

4.5 线程中断方式

- interrupted()
  - 如果一个线程线程处于阻塞、限期等待或者无限期等待状态，那么调用线程的 interrupt() 就会抛出 InterruptedException，从而提前结束该线程

    - 如果一个线程的 run() 方法执行一个无限循环，并且没有执行 sleep() 等会抛出 InterruptedException 的操作，那么调用线程的 interrupt() 方法就无法使线程提前结束，调用 interrupt() 方法会设置线程的中断标记，此时调用 interrupted() 方法会返回 true，因此可以在循环体中使用 interrupted() 方法来判断线程是否处于中断状态，从而提前结束线程
- interrupted() 结束线程
    - 线程处于阻塞状态：如使用了 sleep, 同步锁的 wait, socket 中的 receiver, accept 等方法, 当调用线程的 interrupt()方法时，会抛出 InterruptException 异常
    - 线程未处于阻塞状态：使用 isInterrupted()判断线程的中断标志来退出循环，当使用interrupt()方法时，中断标志就会置 true
- suspend()、resume()和stop()
    - 三个方法已经deprecated
    - suspend()、resume()和stop()方法完成了线程的暂停、恢复和终止工作，但不建议使用 (线程不安全)
    - suspend()方法调用后，线程不会释放已经占有的资源（比如锁），而是占有着资源进入睡眠状态，这样容易引发死锁问题
    - stop()方法在终结一个线程时不会保证线程的资源正常释放，通常是没有给予线程完成资源释放工作的机会，因此会被保护数据就有可能呈现不一致性，其他线程在使用这些被破坏的数据时，有可能导致一些很奇怪的应用程序错误
- Executor 的中断操作
    - 调用 Executor 的 shutdown() 方法会等待线程都执行完毕之后再关闭
    - 但是如果调用的是 shutdownNow() 方法，则相当于调用每个线程的 interrupt() 方法
    - 如果只想中断 Executor 中的一个线程，可以通过使用 submit() 方法来提交一个线程，它会返回一个 Future<?> 对象，通过调用该对象的 cancel(true) 方法就可以中断线程

4.6 线程间通信

- 当多个线程可以一起工作去解决某个问题时，如果某些部分必须在其它部分之前完成，那么就需要对线程进行协调通信
- 在共享内存的并发模型里，线程之间共享程序的公共状态，通过写-读内存中的公共状态进行隐式通信
- **volatile和synchronized关键字**
    - Java支持多个线程同时访问一个对象或者对象的成员变量，由于每个线程可以拥有这个变量的拷贝，所以程序在执行过程中，一个线程看到的变量并不一定是最新的
    - **关键字volatile**可以用来修饰字段（成员变量），就是告知程序任何对该变量的访问均需要从共享内存中获取，而对它的改变必须同步刷新回共享内存，它能保证所有线程对变量访问的可见性
    - 定义一个成员变量 `boolean on=true`, 因为涉及多个线程对变量的访问，因此需要将其定义成为 `volatile boolean on=true`， 这样其他线程对它进行改变时，可以让所有线程感知到变化，因为所有对on变量的访问和修改都需要以共享内存为准
    - **关键字synchronized**可以修饰方法或者以同步块的形式来进行使用，它主要确保多个线程在同一个时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性

    - 对于同步块的实现使用了monitorenter和monitorexit指令，而同步方法则是依靠方法修饰符上的ACC_SYNCHRONIZED来完成的。无论采用哪种方式，其本质是对一个对象的监视器（monitor）进行获取，而这个获取过程是排他的，也就是同一时刻只能有一个线程获取到由synchronized所保护对象的监视器
    - 任意线程对Object（Object由synchronized保护）的访问，首先要获得Object的监视器。如果获取失败，线程进入同步队列，线程状态变为BLOCKED。当访问Object的前驱（获得了锁的线程）释放了锁，则该释放操作唤醒阻塞在同步队列中的线程，使其重新尝试对监视器的获取
- **wait / notify机制**
    - 等待/通知机制是指一个线程A调用了对象O的wait()方法进入等待状态，而另一个线程B调用了对象O的notify()或者notifyAll()方法，线程A收到通知后从对象O的wait()方法返回，进而执行后续操作
    - 调用wait()、notify()和notifyAll()时需要先对调用对象加锁
    - 它们都属于 Object 的方法，只能用在同步方法或者同步控制块中使用，否则会在运行时抛出 IllegalMonitorStateExeception
    - 调用wait()方法后，线程状态由RUNNING变为WAITING，释放锁，并将当前线程放置到对象的等待队列
    - notify()或notifyAll()方法调用后，等待线程依旧不会从wait()返回，需要调用notify()或notifAll()的线程释放锁之后，等待线程才有机会从wait()返回
    - notify()方法将等待队列中的一个等待线程从等待队列中移到同步队列中，而notifyAll()方法则是将等待队列中所有的线程全部移到同步队列，被移动的线程状态由WAITING变为BLOCKED
    - 从wait()方法返回的前提是获得了调用对象的锁
- **await() signal() signalAll()**
    - java.util.concurrent 类库中提供了 Condition 类来实现线程之间的协调，可以在 Condition 上调用 await() 方法使线程等待，其它线程调用 signal() 或 signalAll() 方法唤醒等待的线程
    - 相比 wait() 这种等待方式，await() 可以指定等待的条件，因此更加灵活
    - Condition是个接口，基本的方法就是await()和signal()方法, 调用Condition的await()和signal()方法，都必须在lock保护之内，就是说必须在lock.lock()和lock.unlock之间才可以使用
- wait()和Condition.await()的区别
    - wait()是Object的方法，await()是Condition 类的方法
    - Object.wait()和Condition.await()的原理是基本一致的
    - 不同的是Condition.await()底层是调用LockSupport.park()来实现阻塞当前线程的
- **管道输入/输出流**
    - 管道输入/输出流以内存为传输的媒介，用于线程之间的数据传输
    - PipedOutputStream、PipedInputStream面向字节；PipedReader、PipedWriter面向字符

```
PipedWriter out = new PipedWriter();
PipedReader in = new PipedReader();
// 将输出流和输入流进行连接，否则在使用时会抛出IOException
out.connect(in);
```

- **Thread.join()的使用**

- 在线程中调用另一个线程的 join() 方法，会将当前线程挂起，而不是忙等待，直到目标线程结束
- 当线程终止时，会调用线程自身的notifyAll()方法，会通知所有等待在该线程对象上的线程

- **ThreadLocal的使用**
  - ThreadLocal，即线程变量，是一个以ThreadLocal对象为键、任意对象为值的存储结构
  - 这个结构被附带在线程上，也就是说一个线程可以根据一个ThreadLocal对象查询到绑定在这个线程上的一个值
  - 通过set(T)方法来设置一个值，在当前线程下再通过get()方法获取到原先设置的值

4.7 线程安全定义

- 可以将共享数据按照安全程度的强弱顺序分成以下五类: 不可变、绝对线程安全、相对线程安全、线程兼容和线程对立
- 不可变
  - 不可变(Immutable)的对象一定是线程安全的，不需要再采取任何的线程安全保障措施。只要一个不可变的对象被正确地构建出来，永远也不会看到它在多个线程之中处于不一致的状态
  - 多线程环境下，应当尽量使对象成为不可变，来满足线程安全
  - 不可变的类型: final 关键字修饰的基本数据类型， String，枚举类型，Number 部分子类
- 绝对线程安全
  - 不管运行时环境如何，调用者都不需要任何额外的同步措施
- 相对线程安全
  - 相对线程安全需要保证对这个对象单独的操作是线程安全的，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
- 线程兼容
  - 线程兼容是指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地使用
  - Java API 中大部分的类都是属于线程兼容的，如与前面的 Vector 和 HashTable 相对应的集合类 ArrayList 和 HashMap 等
- 线程对立
  - 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。
  - 由于 Java 语言天生就具备多线程特性，线程对立这种排斥多线程的代码是很少出现的，而且通常都是有害的，应当尽量避免

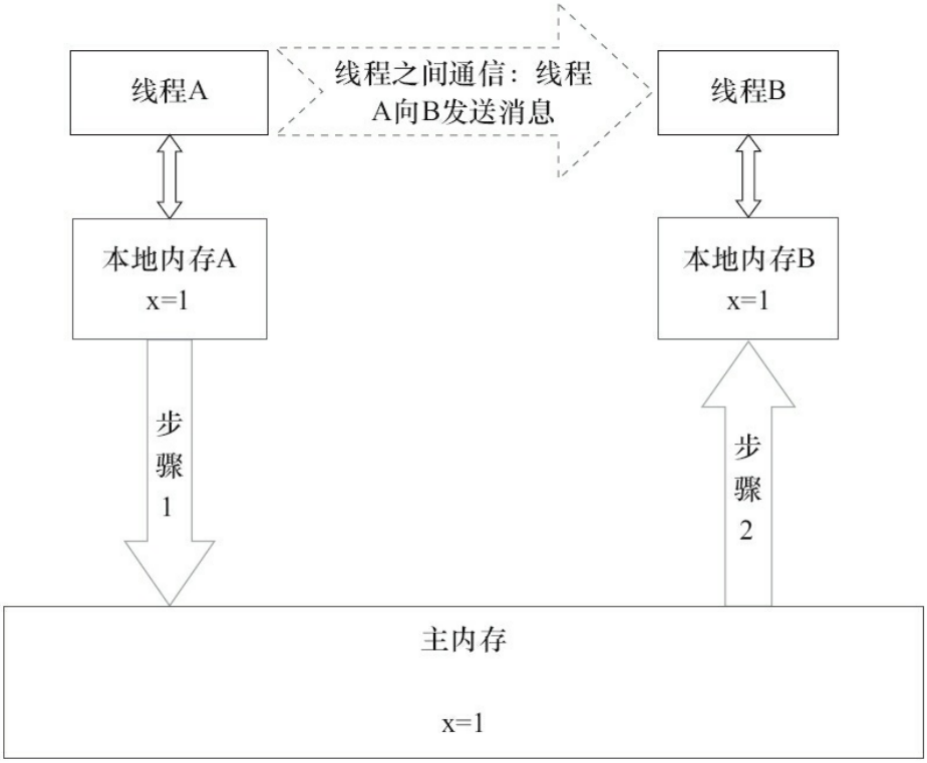
4.8 并发三要素

- **可见性**：多线程并发访问共享变量时候，一个线程对变量的修改能够被其他线程能立马看到
  - 线程A改变了一个变量，变量储存在线程A的高速缓存中，并不会立即更新到内存，以至于其他线程可能会看到脏数据
  - 使用volatile会线程修改变量时同时修改高速缓存和内存的值

- **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
- **有序性**：程序执行的顺序和代码的顺序保持一致
    - JVM在真正执行代码的时候会发生指令重排序（Instruction Reorder）

4.9 Java怎么解决并发问题

- 在Java中，所有实例域、静态域和数组元素都存储在堆内存中，堆内存在线程之间共享。局部变量（Local Variables），方法定义参数（Formal Method Parameters）和异常处理器参数（ExceptionHandler Parameters）不会在线程之间共享，它们不会有内存可见性问题，也不受内存模型的影响
- Java线程之间的通信由 Java内存模型 (JMM) 控制，JMM决定一个线程对共享变量的写入何时对另一个线程可见
    - JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（Main Memory）中，每个线程都有一个私有的本地内存（Local Memory），本地内存 (抽象概念，并不真实存在) 中存储了该线程以读/写共享变量的副本



- 可见性，有序性，原子性
    - Java内存模型只保证了基本读取和赋值是原子性操作，如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。由于synchronized和Lock能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性

```
x = 10;         //语句1: 直接将数值10赋值给x，也就是说线程执行这个语句的会直接将数值10写入到工作内存中
y = x;          //语句2: 包含2个操作，它先要去读取x的值，再将x的值写入工作内存，虽然读取x的值以及 将x的值写入工作内存 这2个操作都是原子性操作，但是合起来就不是原子性操作了。
x++;            //语句3:  x++包括3个操作：读取x的值，进行加1操作，写入新的值。
x = x + 1;      //语句4:  同语句3
```

    - Java提供了volatile关键字来保证可见性。当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性
    - 另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中，因此可以保证可见性
    - 通过volatile关键字来保证一定的"有序性"。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。
- 指令序列的重排序
    - 执行程序时，为了提高性能，编译器和处理器常常会对指令做重排序 (Instruction Reorder)
    - 从Java源代码到最终实际执行的指令序列，会分别经历下面3种重排序:
        - **编译器优化的重排序**：编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序
        - **指令级并行的重排序**：现代处理器采用了指令级并行技术（Instruction-Level Parallelism，ILP）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
        - **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
    - 第一个属于编译器重排序，后两个属于处理器重排序。这些重排序都可能会导致多线程程序出现内存可见性问题
    - 对于编译器重排序，JMM 的编译器重排序规则会禁止特定类型的编译器重排序（不是所有的编译器重排序都要禁止）。
    - 对于处理器重排序，JMM 的处理器重排序规则要求 java 编译器在生成指令序列时，插入特定类型的内存屏障 (memory barriers，intel 称之为 memory fence) 指令，通过内存屏障指令来禁止特定类型的处理器重排序
- volatile 禁止重排序
    - volatile 写是在前面和后面分别插入内存屏障，而 volatile 读操作是在后面插入两个内存屏障
    - LoadLoad 屏障禁止下面所有的普通读操作和上面的 volatile 读重排序
    - LoadStore 屏障禁止下面所有的普通写操作和上面的 volatile 读重排序
    - StoreStore 屏障禁止上面的普通写和下面的 volatile 写重排序, 优先刷新到主内存
    - StoreLoad 屏障防止上面的 volatile 写与下面可能有的 volatile 读/写重排序, 优先刷新到主内存
- JMM(Java内存模型) Happens-Before 规则

- JVM 还规定了先行发生原则，让一个操作无需控制就能先于另一个操作完成
- 单一线程原则 Single Thread rule：在一个线程内，在程序前面的操作先行发生于后面的操作
- 监视器锁规则 Monitor Lock Rule：一个 unlock 操作先行发生于后面对同一个锁的 lock 操作
- volatile变量规则 Volatile Variable Rule：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
- 线程启动规则 Thread Start Rule: Thread 对象的 start() 方法调用先行发生于此线程的每一个动作
- 线程加入规则 Thread Join Rule: Thread 对象的结束先行发生于 join() 方法返回
- 线程中断规则 Thread Interruption Rule: 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事件的发生，可以通过 interrupted() 方法检测到是否有中断发生
- 对象终结规则 Finalizer Rule: 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
- 传递性 Transitivity: 如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那么操作 A 先行发生于操作 C

4.10 Synchronized详解

- synchronized实现同步的基础: Java中的每一个对象都可以作为锁
  - 对于普通同步方法，锁是当前实例对象
  - 对于静态同步方法，锁是当前类的Class对象
  - 对于同步方法块，锁是synchronized括号里配置的对象
- synchronized的使用
  - 一把锁只能同时被一个线程获取，没有获得锁的线程只能等待
  - 每个实例都对应有自己的一把锁(this), 不同实例之间互不影响
  - synchronized修饰的方法，无论方法正常执行完毕还是抛出异常，都会释放锁
  - 对象锁: 包括方法锁(默认锁对象为this,当前实例对象)和同步代码块锁(自己指定锁对象)
  - 类锁: 指synchronized修饰静态的方法或指定锁对象为Class对象
- Synchonized 实现原理
  - JVM基于进入和退出`Monitor`对象来实现方法同步和代码块同步，使用`monitorenter`和`monitorexit`指令实现
  - `monitorenter`指令是在编译后插入到同步代码块的开始位置，而`monitorexit`是插入到方法结束处和异常处
  - 任何对象都有一个`monitor`与之关联，当且一个`monitor`被持有后，它将处于锁定状态
  - 线程执行到`monitorenter`指令时，将会尝试获取对象所对应的`monitor`的所有权，即尝试获得对象的锁
- 加锁和释放锁
  - `Monitorenter`和`monitorexit`指令会让对象在执行时使其锁计数器加1或者减1
  - 在尝试获得`monitor`的所有权时, 会发生如下三种情况之一：
    - monitor计数器为0，意味着monitor还没有被获得，那这个线程就会立刻获得然后把锁计数器+1，此时别的线程再想获取，就需要等待

        - 如果已经拿到这个monitor的所有权，又重入了这把锁，那锁计数器就会累加，变成2，并且随着重入的次数，会一直累加
        - monitor计数器不为0但是其他线程持有，说明这把锁已经被别的线程获取了，等待锁释放
    - `monitorexit`指令就是释放对于`monitor`的所有权，释放过程就是将monitor的计数器减1
        - 如果减完以后，计数器不是0，则代表刚才是重入进来的，当前线程还继续持有这把锁的所有权
        - 如果计数器变成0，则代表当前线程不再拥有该monitor的所有权，即释放锁
- 锁的优化
    - JVM中`monitorenter`和`monitorexit`字节码依赖于底层的操作系统的Mutex Lock来实现的，但是由于使用Mutex Lock需要将当前线程挂起并从用户态切换到内核态来执行，这种切换的代价是非常昂贵的，如果每次都调用Mutex Lock那么将严重的影响程序的性能，jdk1.6中对锁的实现引入了大量的优化来减少锁操作的开销
    - JVM 层面优化
        - `锁粗化(Lock Coarsening)`：也就是减少不必要的紧连在一起的unlock / lock操作, 将多个连续的锁扩展成一个范围更大的锁
        - `锁消除(Lock Elimination)`：通过运行时JIT编译器的逃逸分析来**消除一些不可能存在共享数据竞争的锁。**锁消除的主要判定依据来源于逃逸分析的数据支持。意思就是：JVM会判断再一段程序中的同步明显不会逃逸出去从而被其他线程访问到，那JVM就把它们当作栈上数据对待，认为这些数据是线程独有的，不需要加同步，此时就会进行锁消除
    - 锁一共有4种状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态
        - 锁可以升级但不能降级，目的是为了提高获得锁和释放锁的效率
        - `偏向锁(Biased Locking)`：当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程在进入和退出同步块时不需要进行CAS操作来加锁和解锁，只需测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁。偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。这是为了在无锁竞争的情况下避免在锁获取过程中执行不必要的CAS原子指令
        - `轻量级锁(Lightweight Locking)`：轻量级锁基于这样一种假设，即在真实的情况下我们程序中的大部分同步代码一般都处于无锁竞争状态(即单线程执行环境)，在无锁竞争的情况下完全可以避免调用操作系统层面的重量级互斥锁，取而代之的是在monitorenter和monitorexit中只需要依靠一条CAS原子指令就可以完成锁的获取及释放。加锁阶段 线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁或者进入到阻塞状态。解锁阶段 线程会使用原子的CAS操作将Displaced Mark Word替换回到对象头，如果成功，则表示没有竞争发生。如果失败，表示当前锁存在竞争，锁就会膨胀成重量级锁
        - `适应性自旋(Adaptive Spinning)`：当线程在获取轻量级锁的过程中执行CAS操作失败时，在进入与monitor相关联的操作系统重量级锁(mutex semaphore)前会进入忙等待(Spinning)然后再次尝试，当尝试一定的次数后如果仍然没有成功则调用与该monitor关联的semaphore(即互斥锁)进入到阻塞状态

4.11 Volatile 详解

- 相比Sychronized 重量级锁，volatile是轻量级的synchronized
- volatile保证变量的可见性和有序性，不保证原子性 (单个volatile变量的读/写具有原子性, 复合操作不具有原子性)
- volatile比synchronized的使用和执行成本更低，因为不会引起线程上下文的切换和调度
- volatile 可见性实现
  - 如果一个字段被声明成volatile，Java线程内存模型确保所有线程看到这个变量的值是一致的
  - 对volatile属性的数据进行修改的时候，会引发了两件事情
    - Lock前缀指令会引起处理器缓存写回到系统内存
    - 一个处理器的缓存写回内存的操作会导致其他处理器该数据的缓存无效
  - 如果对声明了volatile的变量进行写操作，JVM就会向处理器发送一条Lock前缀的指令，将这个变量所在缓存行 (cache line)的数据写回到系统内存，对这个变量的读取也会直接从系统内存中读取
- volatile 有序性实现
  - 可见性底层是基于内存屏障(Memory Barrier)实现，内存屏障又称内存栅栏，是一个 CPU 指令
  - 在程序运行时，为了提高执行性能，编译器和处理器会对指令进行重排序，JMM 为了保证在不同的编译器和 CPU 上有相同的结果，通过插入特定类型的内存屏障来禁止+ 特定类型的编译器重排序和处理器重排序，插入一条内存屏障会告诉编译器和 CPU：不管什么指令都不能和这条 Memory Barrier 指令重排序
  - happens-before 规则中有一条是 volatile 变量规则: 对一个 volatile 域的写, happens-before 于任意后续对这个 volatile 域的读

4.12 线程安全的实现方法

- 阻塞 (互斥) 同步
  - **synchronized关键字，JUC下的ReentrantLock**
  - 互斥同步最主要的问题就是线程阻塞和唤醒所带来的性能问题，因此这种同步也称为阻塞同步
  - 互斥同步属于一种悲观的并发策略，总是认为只要不去做正确的同步措施，那就肯定会出现问题。无论共享数据是否真的会出现竞争，它都要进行加锁(这里讨论的是概念模型，实际上虚拟机会优化掉很大一部分不必要的加锁)、用户态核心态转换、维护锁计数器和检查是否有被阻塞的线程需要唤醒等操作
  - synchronized
    - 它只作用于同一个对象，如果调用两个对象上的同步代码块，就不会进行同步
    - **同步一个代码块**: `synchronized (this) {// ...}`
    - **同步一个方法**: `public synchronized void func () {// ...}`
    - **同步一个类**: 作用于整个类，也就是说两个线程调用同一个类的不同对象上的这种同步语句，也会进行同步
  - ReentrantLock
    - ReentrantLock 是 java.util.concurrent(J.U.C)包中的锁

```
public class LockExample {
    private Lock lock = new ReentrantLock();
    public void func() {
        lock.lock();
        try {
            for (int i = 0; i < 10; i++) {
                System.out.print(i + " ");
            }
        } finally {
            lock.unlock(); // 确保释放锁，从而避免发生死锁。
        }
    }
}
```

- 比较
  - **锁的实现**: synchronized 是 JVM 实现的，而 ReentrantLock 是 JDK 实现的
  - **性能**: synchronized 与 ReentrantLock 大致相同，时间代价 volatile<=syn<=lock
  - **等待可中断**: synchronized 不可中断，ReentrantLock 可中断
  - **公平锁**:
    - 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
    - synchronized 中的锁是非公平的，ReentrantLock 默认情况下也是非公平的，但是也可以是公平的
  - **锁绑定多个条件**: 一个 ReentrantLock 可以同时绑定多个 Condition 对象
- 使用选择
  - 除非需要使用 ReentrantLock 的高级功能，否则优先使用 synchronized
  - 这是因为 synchronized 是 JVM 实现的一种锁机制，JVM 原生地支持它，而 ReentrantLock 不是所有的 JDK 版本都支持
  - 并且使用 synchronized 不用担心没有释放锁而导致死锁问题，因为 JVM 会确保锁的释放

- 非阻塞同步
  - 非阻塞同步先进行操作，如果没有其它线程争着用共享数据，那么操作就成功了，如果产生了冲突，那就采取其它的补偿措施
  - **volatile关键字，原子类 (AtomicInteger,AtomicLong)，CAS**
  - CAS
    - 随着硬件指令集的发展，我们可以使用基于冲突检测的乐观并发策略: 先进行操作，如果没有其它线程争用共享数据，那操作就成功了，否则采取补偿措施(不断地重试，直到成功为止)。这种乐观的并发策略的许多实现都不需要将线程阻塞，因此这种同步操作称为非阻塞同步
    - 乐观锁需要操作和冲突检测这两个步骤具备原子性，这里就不能再使用互斥同步来保证了，只能靠硬件来完成。硬件支持的原子性操作最典型的是: 比较并交换(Compare-and-Swap，CAS)。CAS 指令需要有 3 个操作数，分别是内存地址 V、旧的预期值 A 和新值 B。当执行操作时，只有当 V 的值等于 A，才将 V 的值更新为 B
  - AtomicInteger

        - J.U.C 包里面的整数原子类 AtomicInteger，提供原子操作, 其中的 compareAndSet() 和 getAndIncrement() 等方法都使用了 CAS+volatile来实现线程安全的数值操作

- 无同步方案
    - 无同步方案就是从代码逻辑上保证线程安全，如果一个方法本来就不涉及共享数据，那就无须任何同步措施去保证正确性
    - **栈封闭**
        - 多个线程访问同一个方法的局部变量时，不会出现线程安全问题，因为局部变量存储在虚拟机栈中，属于线程私有的
    - **线程本地存储(Thread Local Storage)**
        - 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。如果能保证，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
        - 可以使用 java.lang.ThreadLocal 类来实现线程本地存储功能
        - ThreadLocal 从理论上讲并不是用来解决多线程并发问题的，因为根本不存在多线程竞争
        - 底层数据结构导致 ThreadLocal 有内存泄漏的情况，应该尽可能在每次使用 ThreadLocal 后手动调用 remove()
    - **可重入代码(Reentrant Code)**
        - 这种代码也叫做纯代码(Pure Code)，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
        - 可重入代码有一些共同的特征，例如不依赖存储在堆上的数据和公用的系统资源、用到的状态量都由参数中传入、不调用非可重入的方法等

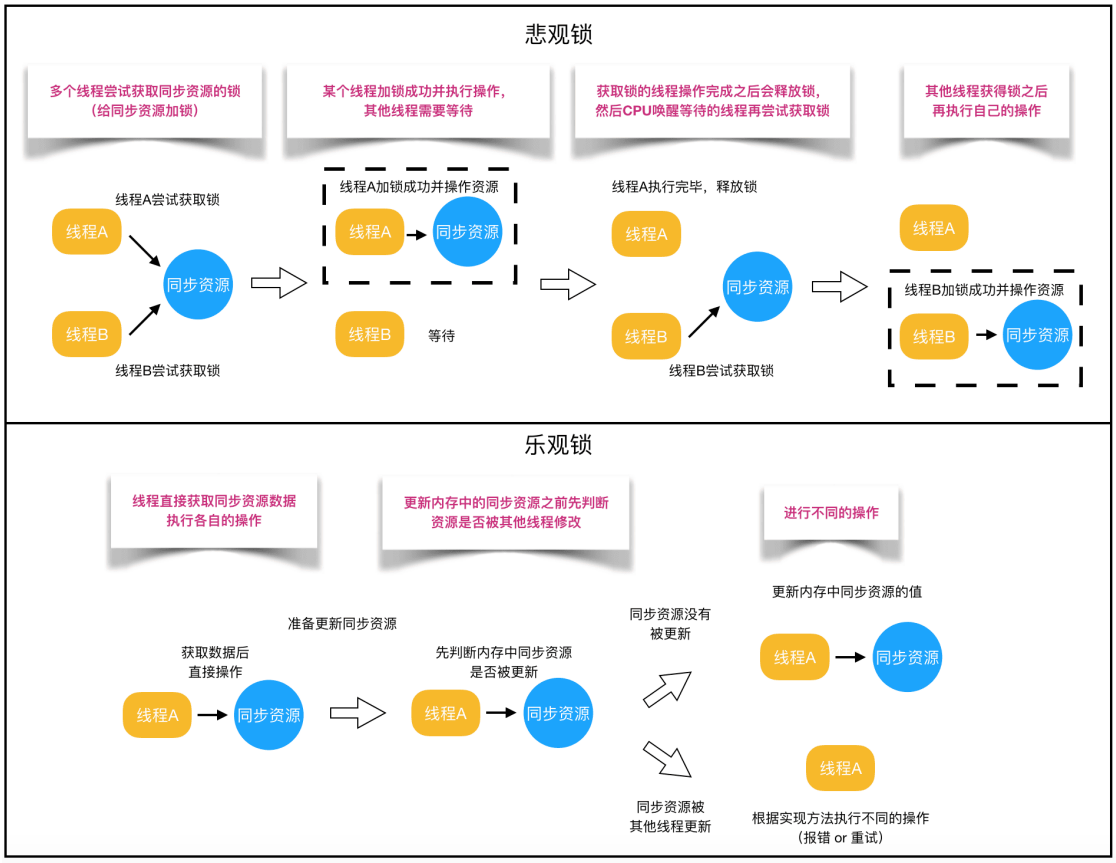
4.13 JUC的结构

- Locks (锁包)：
    - JDK提供的锁机制，相比synchronized关键字来进行同步锁，功能更加强大，它为锁提供了一个框架
    - LockSupport 它具备阻塞线程和解除阻塞线程的功能，并且不会引发死锁
    - ReentrantLock 它是独占锁，是指只能被独自占领，即同一个时间点只能被一个线程锁获取到的锁
    - ReadWriteLock 它包括子类ReadLock和WriteLock。ReadLock是共享锁，而WriteLock是独占锁
- tools (工具类)：又叫信号量三组工具类
    - CountDownLatch（闭锁） 是一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待
    - CyclicBarrier（栅栏） 之所以叫barrier，是一个同步辅助类，允许一组线程互相等待，直到到达某个公共屏障点 ，并且在释放等待线程后可以重用
    - Semaphore（信号量） 是一个计数信号量，它的本质是一个“共享锁“。信号量维护了一个信号量许可集。线程可以通过调用 acquire()来获取信号量的许可；当信号量中有可用的许可时，线程能获取该许可；否则线程必须等待，直到有可用的许可为止。 线程可以通过release()来释放它所持有的信号量许可

- Executor (执行者)：是Java里面线程池的顶级接口，但它只是一个执行线程的工具，真正的线程池接口是ExecutorService
  - https://blog.csdn.net/weixin_43888181/article/details/116518664?spm=1001.2014.3001.5501
  - ScheduledExecutorService 解决那些需要任务重复执行的问题
  - ScheduledThreadPoolExecutor 周期性任务调度的类实现
- Atomic(原子性包)：是JDK提供的一组原子操作类
  - 基本原子变量类
    - 实现原理大多是持有它们各自的对应的类型变量value，而且被volatile关键字修饰了
    - AtomicBoolean
    - AtomicInteger
    - AtomicIntegerArray
  - Array
- Collections(集合类)：主要是提供线程安全的集合
  - HashMap对应的高并发类是ConcurrentHashMap
  - ArrayList对应的高并发类是CopyOnWriteArrayList
  - HashSet对应的高并发类是 CopyOnWriteArraySet

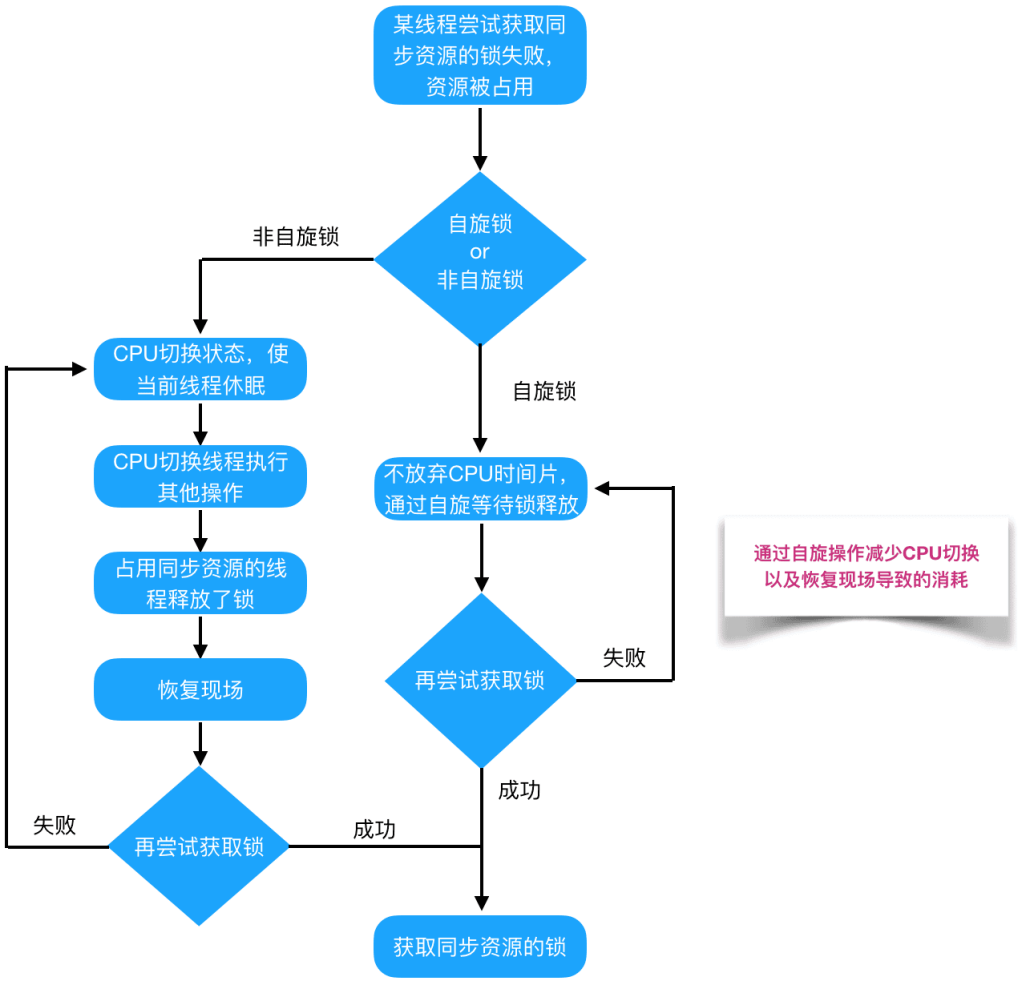
4.14 Java中的锁

- 乐观锁 VS 悲观锁
  - 判断：要不要锁住同步资源



- 悲观锁认为在使用数据的时候一定有别的线程来修改数据，因此在获取数据的时候会先加锁，确保数据不会被别的线程修改
- 乐观锁认为在使用数据时不会有别的线程修改数据，所以不会添加锁，只是在更新数据的时候去判断之前有没有别的线程更新了这个数据。如果这个数据没有被更新，当前线程将自己修改的数据成功写入。如果数据已经被其他线程更新，则根据不同的实现方式执行不同的操作（例如报错或者自动重试）
- 乐观锁在Java中是通过使用无锁编程来实现，最常采用的是CAS算法，Java原子类中的递增操作就通过CAS自旋实现的
- 适合场景
  - 悲观锁适合写操作多的场景，先加锁可以保证写操作时数据正确。
  - 乐观锁适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升

- 自旋锁 VS 适应性自旋锁
  - **自旋锁**就是线程在获取锁的过程中，不会去阻塞线程，也就无所谓唤醒线程



  - 阻塞或唤醒一个Java线程需要操作系统切换CPU状态来完成，这种状态转换需要耗费处理器时间。在许多场景中，同步资源的锁定时间很短，为了这一小段时间去切换线程，线程挂起和恢复现场的花费可能会让系统得不偿失。如果物理机器有多个处理器，能够让两个或以上的线程同时并行执行，我们就可以让后面那个请求锁的线程不放弃CPU的执行时间，看看持有锁的线程是否很快就会释放锁。而为了让当前线程“稍等一下”，我们需让当前线程进行自旋，如果在自旋完成后前面锁定同步资源的线程已经释放了锁，那么当前线程就可以不必阻塞而是直接获取同步资源，从而避免切换线程的开销, 这就是自旋锁。
  - 自旋锁本身是有缺点的，它不能代替阻塞。自旋等待虽然避免了线程切换的开销，但它要占用处理器时间。如果锁被占用的时间很短，自旋等待的效果就会非常好。反之，如果锁被占用的时间很长，那么自旋的线程只会白浪费处理器资源。
  - 自旋等待的时间必须要有一定的限度，如果自旋超过了限定次数（默认是10次，可以使用-XX:PreBlockSpin来更改）没有成功获得锁，就应当挂起线程
  - **自适应自旋锁**意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定的。如果在同一个锁对象上，自旋等待刚刚成功获取过锁，并且持有锁的线程正在运行中，那么JVM会认为该锁自旋获取到锁的可能性很大，会自动增加等待时间。比如增加到100此循环。相反，如果对于某个锁，自旋很少成功获取锁。那再以后要获取这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。有了自适应自旋，JVM对程序的锁的状态预测会越来越准确，JVM也会越来越聪明
- 偏向锁 VS 轻量级锁 VS 重量级锁
  - 这四种锁是指锁的状态，专门针对synchronized的

    - 偏向锁：当一段代码没有别的线程访问，此时线程去访问会直接获取偏向锁
        - 在锁对象的对象头中记录一下当前获取到该锁的线程ID，如果再来就可以直接获取到
        - 偏向锁通过对比Mark Word解决加锁问题，避免执行CAS操作
    - 轻量级锁：当锁是偏向锁时，有另外一个线程来访问，会升级为轻量级锁。
        - 线程会通过CAS方式和自旋获取锁，并不会阻塞，提高性能
    - 重量级: 轻量级锁自旋一段时间后线程还没有获取到锁，会升级为重量级锁
        - 重量级锁时，来竞争锁的所有线程都会阻塞，性能降低
    - 注意，锁只能升级不能降级
- 公平锁 VS 非公平锁
    - 判断：多个锁竞争要不要排队
    - **公平锁**是指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获得锁。
    - 公平锁的优点是等待锁的线程不会饿死
    - 缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU唤醒阻塞线程的开销比非公平锁大
    - **非公平锁**是多个线程加锁时直接尝试获取锁，获取不到才会到等待队列的队尾等待。但如果此时锁刚好可用，那么这个线程可以无需阻塞直接获取到锁，所以非公平锁有可能出现后申请锁的线程先获取锁的场景
    - 非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU不必唤醒所有线程
    - 缺点是处于等待队列中的线程可能会饿死，或者等很久才会获得锁
- 可重入锁 VS 非可重入锁
    - 判断：一个线程的多个流程能不能重复获得同一把锁
    - 可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁（前提锁对象得是同一个对象或者class），不会因为之前已经获取过还没释放而阻塞。
    - Java中ReentrantLock和synchronized都是可重入锁，可重入锁的一个优点是可一定程度避免死锁。
    - 非可重入锁在重复调用同步资源时会出现死锁
- 独享锁(排他锁) VS 共享锁
    - 判断：多个线程能不能共享一把锁
    - **独享锁也叫排他锁**，是指该锁一次只能被一个线程所持有。如果线程T对数据A加上排它锁后，则其他线程不能再对A加任何类型的锁。获得排它锁的线程即能读数据又能修改数据。
    - 共享锁是指该锁可被多个线程所持有。如果线程T对数据A加上共享锁后，则其他线程只能对A再加共享锁，不能加排它锁。获得共享锁的线程只能读数据，不能修改数据。
    - 独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享

4.15 JUC Locks LockSupport

- 锁是用来控制多个线程访问共享资源的方式，一般一个锁能够防止多个线程同时访问共享资源 (允许多个线程并发访问的读写锁除外)
- Java 1.5之后JUC新增Lock接口提供了与synchronized关键字类似的同步功能，只是在使用时需要显式地获取和释放锁
  - 使用synchronized关键字将会隐式地获取锁，但是它将锁的获取和释放固化了，也就是先获取再释放
  - Lock接口拥有了锁获取与释放的可操作性、可中断的获取锁以及超时获取锁等同步特性
- LockSupport用来创建锁和其他同步类的基本线程阻塞原语，提供了最基本的线程阻塞和唤醒功能
  - LockSupport位置是`java.util.concurrent.locks.LockSupport`
- LockSupport的核心函数都是基于sun.misc.Unsafe类中的park和unpark函数
  - park函数，阻塞线程，并且该线程在下列情况发生之前都会被阻塞
    - ① 调用unpark函数，释放该线程的许可。② 该线程被中断。③ 设置的时间到了
    - 当time为绝对时间时，isAbsolute为true，否则，isAbsolute为false。当time为0时，表示无限等待，直到unpark发生
    - `public native void park(boolean isAbsolute, long time);`
  - unpark函数，释放线程的许可，即激活调用park后阻塞的线程。这个函数不是安全的，调用这个函数时要确保线程依旧存活
- 核心函数
  - 调用LockSupport.park时，表示当前线程将会等待，直至获得许可
    - 调用了park函数后，线程都将处于休眠状态直至被调用 unpark或者当前线程被中断

```
public static void park() {
    UNSAFE.park(false, 0L); // 调用Unsafe的park函数，获取许可，设置时间为无限长，直到可以获取许可
}
```

  - 调用LockSupport.unpark时，必须把等待获得许可的线程作为参数进行传递，让此线程继续运行
    - 线程为不空, 释放该线程许可

```
public static void unpark(Thread thread) {
    if (thread != null) // 线程为不空
        UNSAFE.unpark(thread); // 释放该线程许可
}
```

4.16 JUC Locks AQS

- 什么是AQS?
  - 队列同步器 AbstractQueuedSynchronizer 是用来构建锁或者其他同步组件的基础框架，它使用了一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作

    - AQS主要使用方式是继承，子类通过继承AQS并实现它的抽象方法来管理同步状态，在抽象方法的实现过程中免不了要对同步状态进行更改，这时就需要使用同步器提供的3个方法:
        - `getState()`：获取当前同步状态
        - `setState(int newState)`：设置当前同步状态
        - `compareAndSetState(int expect,int update)`：使用CAS设置当前状态，该方法能够保证状态设置的原子性
    - AQS内部有三个核心组件，一个是state代表加锁状态初始值为0，一个是获取到锁的线程，还有一个阻塞队列
        - 当有线程想获取锁时，会以CAS的形式将state变为1，CAS成功后便将加锁线程设为自己
        - 当其他线程来竞争锁时会判断state是不是0，不是0再判断加锁线程是不是自己，不是的话就把自己放入阻塞队列，这个阻塞队列是用双向链表实现的
        - 可重入锁的原理就是每次加锁时判断一下加锁线程是不是自己，是的话state+1，释放锁的时候就将state-1。当state减到0的时候就去唤醒阻塞队列的第一个线程.
- AQS的核心思想
    - 如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态
    - 如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中
        - CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列 (仅存在结点之间的关联关系)
    - AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点(Node)来实现锁的分配
    - AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作实现对其值的修改

      ```
      private volatile int state;//共享变量，使用volatile修饰保证线程可见性
      ```

    - 状态信息通过procted类型的`getState`，`setState`，`compareAndSetState`进行操作

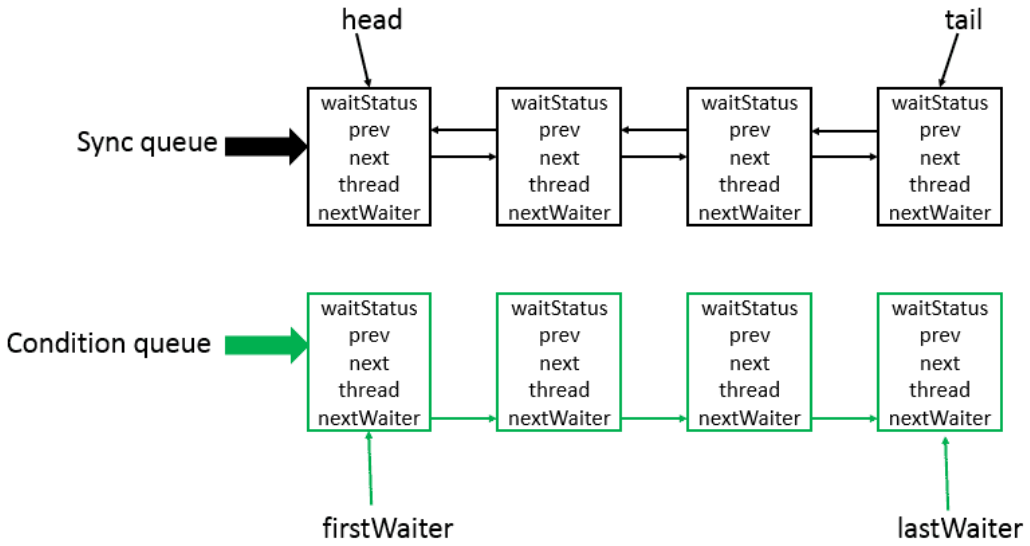
      ```
      //返回同步状态的当前值
      protected final int getState() {
              return state;
      }
       // 设置同步状态的值
      protected final void setState(int newState) {
              state = newState;
      }
      //原子地(CAS操作)将同步状态值设置为给定值update如果当前同步状态的值等于expect(期望值)
      protected final boolean compareAndSetState(int expect, int update) {
              return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
      }
      ```

- AQS定义两种资源共享方式
    - Exclusive(独占): 只有一个线程能访问执行，又根据是否按队列的顺序分为公平锁和非公平锁
        - 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁

        - 非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的
    - Share(共享): 多个线程可同时访问执行，如Semaphore、CountDownLatch
    - ReentrantReadWriteLock可以看成是组合式，允许多个线程同时对某一资源进行读
- AQS底层使用了模板方法模式
    - 使用者继承AbstractQueuedSynchronizer并重写指定的方法, 将AQS组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法

```
isHeldExclusively() //该线程是否正在独占资源 只有用到condition才需要去实现它
tryAcquire(int) //独占方式。尝试获取资源，成功则返回true，失败则返回false
tryRelease(int) //独占方式。尝试释放资源，成功则返回true，失败则返回false
tryAcquireShared(int)//共享方式 尝试获取资源 负数表示失败;0表示成功 但没有剩余可用资源;正数表示成功 且有剩余资源
tryReleaseShared(int)//共享方式 尝试释放资源 成功则返回true;失败则返回false
```

    - AbstractQueuedSynchronizer类底层的数据结构是使用CLH(Craig,Landin,and Hagersten)队列
        - AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点(Node)来实现锁的分配
    - 同步队列 Sync queue 是双向链表，包括head结点和tail结点，head结点主要用作后续的调度
    - 而 Condition queue不是必须的，其是一个单向链表，只有当使用Condition时，才会存在此单向链表。并且可能会有多个Condition queue



4.17 JUC Locks AQS源码

- AQS两个内部类
    - Node类
    - ConditionObject类
- AQS类属性
    - 属性中包含了头节点head，尾结点tail，状态state、自旋时间spinForTimeoutThreshold

```
// 头节点
private transient volatile Node head;
// 尾结点
private transient volatile Node tail;
// 状态
private volatile int state;
// 自旋时间
static final long spinForTimeoutThreshold = 1000L;
```

  - 还有AbstractQueuedSynchronizer抽象的属性在内存中的偏移地址，通过该偏移地址，可以获取和设置该属性的值，同时还包括一个静态初始化块，用于加载内存偏移地址

- 类的核心方法
  - acquire方法: 以独占模式获取资源，忽略中断，即线程在aquire过程中，中断此线程是无效的

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

    - 首先调用`tryAcquire`方法，试图在独占模式下获取对象状态，需要子类去重写此方法
    - 若`tryAcquire`失败，则调用`addWaiter`方法，功能是将调用此方法的线程封装成为一个结点并放入Sync queue
    - 调用`acquireQueued`方法，功能是Sync queue中的结点不断尝试获取资源，若成功，则返回true，否则，返回false
  - release方法: 独占模式释放对象

```
public final boolean release(int arg) {
    if (tryRelease(arg)) { // 释放成功
        // 保存头节点
        Node h = head;
        if (h != null && h.waitStatus != 0) // 头节点不为空并且头节点状态不为0
            unparkSuccessor(h); //释放头节点的后继结点
        return true;
    }
    return false;
}
```

    - 首先调用`tryRelease`方法，需要子类去重写此方法
    - 如果`tryRelease`成功，那么如果头节点不为空并且头节点的状态不为0，则释放头节点的后继结点

- Sync queue 总结
  - 每一个结点都是由前一个结点唤醒
  - 当结点发现前驱结点是head并且尝试获取成功，则会轮到该线程运行
  - condition queue中的结点向sync queue中转移是通过signal操作完成的

    - 当结点的状态为SIGNAL时，表示后面的结点需要运行
- AQS可重写的方法
    - protected boolean tryAcquire(int arg) 独占式获取同步状态，实现该方法需要查询当前状态并判断同步状态是否符合预期，然后再进行 CAS 设置同步状态

```
// 当状态为0的时候获取锁
public boolean tryAcquire(int acquires) {
    if (compareAndSetState(0, 1)) {
        setExclusiveOwnerThread(Thread.currentThread());
        return true;
    }
    return false;
}
```

    - protected boolean tryRelease(int arg) 独占式释放同步状态，等待获取同步状态的线程将有机会获取同步状态

```
// 释放锁，将状态设置为0
protected boolean tryRelease(int releases) {
    if (getState() == 0) throw new
    IllegalMonitorStateException();
    setExclusiveOwnerThread(null);
    setState(0);
    return true;
}
```

    - protected int tryAcquireShared(int arg) 共享式获取同步状态，返回大于等于0的值，表示获取成功，反之，获取失败
    - protected boolean tryReleaseShared(int arg) 共享式释放同步状态
    - protected boolean isHeldExclusively() 当前同步器是否在独占模式下被线程占用，一般该方法表示是否被当前线程所独占

```
// 是否处于占用状态
protected boolean isHeldExclusively() {
    return getState() == 1;
}
```

4.18 JUC Locks ReentrantLock

- ReentrantLock 重入锁表示该锁能够支持一个线程对资源的重复加锁。除此之外，该锁的还支持获取锁时的公平和非公平性选择
- ReentrantLock实现了Lock接口，Lock接口中定义了lock与unlock相关操作，并且还存在newCondition方法，表示生成一个条件
- ReentrantLock类内部总共存在Sync、NonfairSync、FairSync三个类
- Sync类继承自AbstractQueuedSynchronizer抽象类

```
abstract static class Sync extends AbstractQueuedSynchronizer
```

- NonfairSync与FairSync类继承自Sync类
  - NonfairSync类表示采用非公平策略获取锁, 先用CAS抢线程，否则使用acquire公平获取锁

    ```
    static final class NonfairSync extends Sync
    ```

  - FairSync类表示采用公平策略获取锁， 直接使用acquire公平获取锁

    ```
    static final class FairSync extends Sync
    ```

- ReentrantLock类
  - 类的属性
    - ReentrantLock类的sync非常重要，对ReentrantLock类的操作大部分都直接转化为对Sync和AQS类的操作

      ```
      public class ReentrantLock implements Lock, java.io.Serializable
      ```

  - 类的构造函数
    - 默认是采用的**非公平策略**获取锁，或者传递参数确定采用**公平策略或者是非公平策略**

      ```
      public ReentrantLock() {
          // 默认非公平策略
          sync = new NonfairSync();
      }
      public ReentrantLock(boolean fair) {
          sync = fair ? new FairSync() : new NonfairSync();
      }
      ```

  - 核心函数 lock() 和 tryLock()
    - lock()表示阻塞加锁，线程会阻塞直到加到锁，方法也没有返回值
    - tryLock()表示尝试加锁，可能加到，也可能加不到，该方法不会阻塞线程，如果加到锁则返回true，没有加到则返回false

      ```
      Lock lock = new ReentrantLock();
      lock.lock();
      try { method body }
      finally {
          lock.unlock();
      }
      ```

- ReentrantLock详解

4.19 JUC Locks ReadWriteLock

- `ReentrantReadWriteLock`表示可重入读写锁
  - 读写锁在同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞
  - `ReentrantReadWriteLock`中包含了两种锁，读锁`ReadLock`和写锁`WriteLock`
  - 在读多于写的情况下，读写锁能够提供比排它锁更好的并发性和吞吐量
- `ReentrantReadWriteLock`三个特性
  - 公平性选择：支持非公平(默认)和公平的锁获取方式，吞吐量还是非公平优于公平
  - 重进入：该锁支持重进入，以读写线程为例:读线程在获取了读锁之后，能够再次获取读锁。而写线程在获取了写锁之后能够再次获取写锁，同时也可以获取读锁
  - 锁降级：遵循获取写锁、获取读锁再释放写锁的次序，写锁能够降级成为读锁
- ReentrantReadWriteLock底层实现原理
  - ReentrantReadWriteLock实现了ReadWriteLock接口，ReadWriteLock接口定义了获取读锁和写锁的规范
  - 同时其还实现了Serializable接口，表示可以进行序列化
  - ReentrantReadWriteLock属性包括了一个ReentrantReadWriteLock.ReadLock对象，表示读锁；一个ReentrantReadWriteLock.WriteLock对象，表示写锁；一个Sync对象，表示同步队列
- 读写锁实现
  - 读写状态的设计
    - 读写锁的读写状态就是AQS的同步状态，读写锁需要在同步状态（一个整型变量）上维护多个读线程和一个写线程的状态
    - “按位切割使用”，读写锁将变量切分成了两个部分，高16位表示读锁，低16位表示写锁
    - 最大数量是2^16
  - 写锁的获取与释放
    - 如果当前线程已经获取了写锁，则增加写状态
    - 如果当前线程在获取写锁时，读锁已经被获取或者该线程不是已经获取写锁的线程，则当前线程进入等待状态
    - tryAcquire(int acquires)
    - 写锁的释放与ReentrantLock的释放过程基本类似，每次释放均减少写状态，当写状态为0时表示写锁已被释放，从而等待的读写线程能够继续访问读写锁，同时前次写线程的修改对后续读写线程可见
  - 读锁的获取与释放
    - 读锁是一个支持重进入的共享锁，它能够被多个线程同时获取，在没有其他写线程访问，读锁总会被成功地获取，而所做的也只是（线程安全的）增加读状态
    - 如果当前线程已经获取了读锁，则增加读状态
    - 如果当前线程在获取读锁时，写锁已被其他线程获取，则进入等待状态
    - tryAcquireShared(int unused)

- 什么是锁的升降级?
  - 锁降级指的是写锁降级成为读锁
  - 如果当前线程拥有写锁，然后将其释放，最后再获取读锁，这种分段完成的过程不能称之为锁降级
  - 锁降级是指把持住(当前拥有的)写锁，再获取到读锁，随后释放(先前拥有的)写锁的过程
  - 是否必要呢: 保证数据的可见性，如果当前线程不获取读锁而是直接释放写锁, 假设此刻另一个线程(记作线程T)获取了写锁并修改了数据，那么当前线程无法感知线程T的数据更新。如果当前线程获取读锁，即遵循锁降级的步骤，则线程T将会被阻塞，直到当前线程使用数据并释放读锁之后，线程T才能获取写锁进行数据更新
- 为什么不支持锁升级?
  - 目的也是保证数据可见性
  - 如果读锁已被多个线程获取，其中任意线程成功获取了写锁并更新了数据，则其更新对其他获取到读锁的线程是不可见的

4.20 JUC ConcurrentHashMap

- 为什么使用ConcurrentHashMap
  - 线程不安全的HashMap
  - 效率低下的HashTable
    - HashTable容器使用synchronized来保证线程安全，当一个线程访问HashTable，其他线程会进入阻塞或轮询状态
    - 不能使用put方法添加元素，也不能使用get方法来获取元素
  - ConcurrentHashMap的锁分段技术可有效提升并发访问率
    - ConcurrentHashMap 将数据分成一段一段地存储，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
    - `ConcurrentHashMap JDK1.7`: 使用分段锁机制实现;
    - `ConcurrentHashMap JDK1.8`: 使用数组+链表+红黑树数据结构和CAS原子操作实现;
- Java8 数据结构
  - 结构上和 Java8 的 HashMap 基本上一样，但要保证线程安全性，所以源码更复杂
  - 用CAS算法实现无锁化的修改值的操作，他可以大大降低锁的性能消耗
  - 使用mappingCount()而不是size()，因为size只能返回整数，但有可能总数为long
  - 构造函数
    - 计算sizeCtl为 (1.5 * initialCapacity + 1) 向上取最近的 2 的 n 次方

```java
public ConcurrentHashMap(int initialCapacity) {
    if (initialCapacity < 0) throw new IllegalArgumentException();
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
               MAXIMUM_CAPACITY :
               tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
    this.sizeCtl = cap;
}
```

- 初始化数组: initTable
  - 初始化一个合适大小的数组，然后会设置 sizeCtl

```java
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0){
        if ((sc = sizeCtl) < 0) Thread.yield(); // 被其他线程抢先初始化
         // CAS将 sizeCtl 设置为 -1，代表抢到了锁
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)){
            try {
                if ((tab = table) == null || tab.length == 0) {
                    // DEFAULT_CAPACITY 默认初始容量是 16
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    // 初始化数组，长度为 16 或 DEFAULT_CAPACITY
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    // 将这个数组赋值给 volatile table
                    table = tab = nt;
                    // 0.75 * n
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

- Put操作: CAS+synchronized 实现并发插入或更新操作
  - 如果 key || value 为null，会抛出异常 NullPointerException
  - 如果如果数组"空"，进行数组初始化
  - `if` 找该 hash 值对应的数组下标，得到第一个节点 `f`
    - 如果数组待插入的元素位置为空，用CAS 操作将这个新值放入
    - 如果 CAS 失败，那就是有并发操作，进到下一个循环
  - `if else` 如果正在扩容，则当前线程一起加入到扩容的过程中帮助数据迁移 helpTransfer()
  - `else` 如果数组待插入的元素位置不为空，则synchronized 锁住这个头节点（分段锁）

        - 如果头节点 hash值 >= 0，说明以链表方式存储
        - 遍历链表，发现相等的 key，判断是否要进行值覆盖，遍历到链表末端，就将新值放到链表
        - 如果 instance 是 TreeBin，说明以以红黑树方式存储，调用红黑树的插值方法插入新节点
    - 判断是否要将链表转换为红黑树，临界值和 HashMap 一样，也是 8
    - 如果元素存在，则返回旧值；
    - 如果元素不存在，整个Map的元素个数加1，并检查是否需要扩容
    - Get操作
        - get操作全程不需要加锁是因为Node的成员val是用volatile修饰的，和数组用volatile修饰没有关系
        - 计算 hash 值，根据 hash 值找到数组对应位置: (n - 1) & h
        - 根据该位置处结点性质进行相应查找
            - 如果该位置为 null，那么直接返回 null 就可以了
            - 如果该位置处的节点刚好就是我们需要的，返回该节点的值即可
            - 如果该位置节点的 hash 值小于 0，说明正在扩容，或者是红黑树，后面我们再介绍 find 方法
            - 如果以上 3 条都不满足，那就是链表，进行遍历比对即可
- ConcurrentHashMap 扩容机制
    - 1.7版本
        - 1.7版本的ConcurrentHashMap是基于Segment分段实现的
        - 每个Segment相对于一个小型的HashMap
        - 每个Segment内部会进行扩容，和HashMap的扩容逻辑类似
        - 先生成新的数组，然后转移元素到新数组中
        - 扩容的判断也是每个Segment内部单独判断的，判断是否超过阈值
    - 1.8版本
        - 1.8版本的ConcurrentHashMap不再基于Segment实现
        - 如果某个线程put时，发现没有正在进行扩容，则将key-value添加到ConcurrentHashMap中，然后判断是否超过阈值，超过了则进行扩容
        - 某个线程想增/删元素时，如果访问的桶是ForwardingNode节点，则表明当前正处于扩容状态，**协助一起扩容完成后**再完成相应的数据更改操作
        - 扩容时将原table的所有桶倒序分配，每个线程每次最小分16个桶进行处理，防止资源竞争导致的效率下降， 每个桶的迁移是单线程的，但桶范围处理分配可以多线程，在没有迁移完成所有桶之前每个线程需要重复获取迁移桶范围，直至所有桶迁移完成
        - 迁移过程中sizeCtl用于记录参与扩容线程的数量，全部迁移完成后sizeCtl更新为新table的扩容阈值
- [java并发之ConcurrentHashMap 1.8原理详解]

4.21 JUC ConcurrentLinkedQueue

- 实现一个线程安全的队列有两种方式：一种是使用阻塞算法，另一种是使用非阻塞算法
  - 阻塞算法的队列可以用一个锁（入队和出队用同一把锁）或两个锁（入队和出队用不同的锁）等方式来实现
  - 非阻塞的实现方式则可以使用循环CAS的方式来实现
  - ConcurrentLinkedQueue使用非阻塞的方式来实现
- 结构
  - ConcurrentLinkedQueue 继承自AbstractQueue，实现Node
  - ConcurrentLinkedQueue由head节点和tail节点组成，每个 Node 由节点元素 (item) 和指向下一个节点 (next) 的引用组成
  - 默认情况下head节点存储的元素为空，tail节点等于head节点
- 入队列
  - 入队列就是将入队节点添加到队列的尾部，将入队节点设置成当前队列尾节点的下一个节点
  - 如果tail节点的next节点不为空，则将入队节点设置成tail节点，如果tail节点的next节点为空，则将入队节点设置成tail的next节点，所以tail节点不总是尾节点
  - 尾节点可能是tail节点，也可能是tail节点的next节点
- 出队列
  - 出队列的就是从队列里返回一个节点元素，并清空该节点对元素的引用
  - 并不是每次出队时都更新head节点，当head节点里有元素时，直接弹出head节点里的元素，而不会更新head节点；只有当head节点里没有元素时，出队操作才会更新head节点
  - 获取头节点的元素，然后判断头节点元素是否为空，如果为空，表示另外一个线程已经进行了一次出队操作将该节点的元素取走，如果不为空，则使用CAS的方式将头节点的引用设置成null，如果CAS成功，则直接返回头节点的元素，如果不成功，表示另外一个线程已经进行了一次出队操作更新了head节点，导致元素发生了变化，需要重新获取头节点

4.22 JUC CopyOnWriteArrayList

- CopyOnWriteArrayList实现了List接口，List接口定义了对列表的基本操作；同时实现了RandomAccess接口，表示可以随机访问(数组具有随机访问的特性)；同时实现了Cloneable接口，表示可克隆；同时也实现了Serializable接口，表示可被序列化
- 底层原理
  - 因为内部是用数组来实现的，在向CopyOnWriteArrayList添加元素时，会复制一个新的数组，写操作在新数组上进行，读操作在原数组上进行
  - 写操作会加锁，防止出现并发写入丢失数据的问题
  - 写操作结束之后会把原数组指向新数组
- 缺陷和使用场景
  - 由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致young gc或者full gc

- 不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个set操作后，读取到数据可能还是旧的,虽然CopyOnWriteArrayList 能做到最终一致性,但是还是**没法满足实时性**要求
- CopyOnWriteArrayList允许在写操作时来读取数据，大大提高了读的性能，因此适合读多写少的应用场景
  - 不过慎用，没法保证要放置多少数据，万一数据稍微多，每次add/set都要重新复制数组，这个代价实在太高昂

4.23 JUC Java的阻塞队列

- 什么是阻塞队列
  - 阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作支持阻塞的插入和移除方法
  - 当队列中没有数据的情况下，消费者端的所有线程都会被自动阻塞（挂起），直到有数据放入队列
  - 当队列中填满数据的情况下，生产者端的所有线程都会被自动阻塞（挂起），直到队列中有空的位置，线程被自动唤醒
  - 阻塞队列常用于生产者和消费者的场景，生产者是向队列里添加元素的线程，消费者是从队列里取元素的线程。阻塞队列就是生产者用来存放元素、消费者用来获取元素的容器
- 阻塞队列的主要方法，各有4种处理方式（抛出异常、特殊值、阻塞、超时）
  - 插入方法：add(e)，offer(e)，put(e)，offer(e,time,unit)
  - 移除方法：remove()，poll()，take()，poll(time,unit)
  - 检查方法：element()，peek()
- Java里的阻塞队列
  - ArrayBlockingQueue ： 由数组结构组成的有界阻塞队列
  - LinkedBlockingQueue ： 由链表结构组成的有界阻塞队列
  - PriorityBlockingQueue ： 支持优先级排序的无界阻塞队列
  - DelayQueue：使用优先级队列实现的无界阻塞队列
  - SynchronousQueue：不存储元素的阻塞队列
  - LinkedTransferQueue：由链表结构组成的无界阻塞队列
  - LinkedBlockingDeque：由链表结构组成的双向阻塞队列
- ArrayBlockingQueue（公平、非公平）
  - ArrayBlockingQueue是一个用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序
  - 默认情况下不保证线程公平的访问队列，所谓公平访问队列是指阻塞的线程，可以按照阻塞的先后顺序访问队列，即先阻塞线程先访问队列
  - 非公平性是对先等待的线程是非公平的，当队列可用时，阻塞的线程都可以争夺访问队列的资格，有可能先阻塞的线程最后才访问队列

```
ArrayBlockingQueue fairQueue = new ArrayBlockingQueue(1000,true); //公平的
```

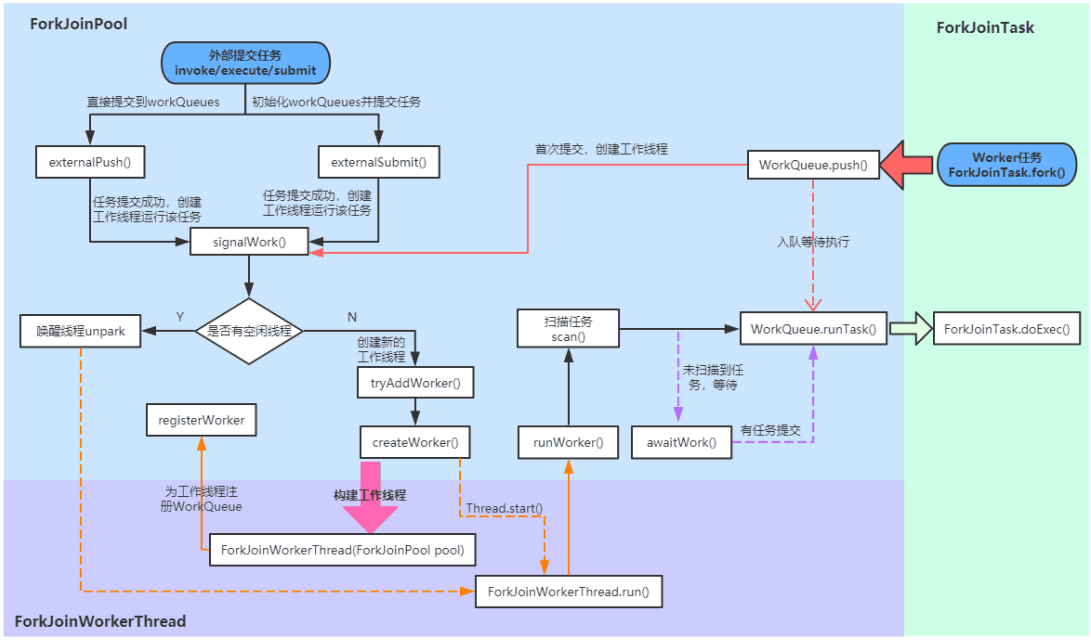
- LinkedBlockingQueue（两个独立锁提高并发）
  - 基于链表的阻塞队列，同 ArrayListBlockingQueue 类似，此队列按照先进先出（FIFO）的原则对元素进行排序
  - LinkedBlockingQueue 之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能
  - LinkedBlockingQueue 会默认一个类似无限大小的容量（Integer.MAX_VALUE）
- PriorityBlockingQueue（compareTo 排序实现优先）
  - 支持优先级的无界队列。默认情况下元素采取自然顺序升序排列
  - 可以自定义实现compareTo()方法来指定元素进行排序规则，或者初始化 PriorityBlockingQueue 时，指定构造参数 Comparator 来对元素进行排序
  - 需要注意的是不能保证同优先级元素的顺序
- DelayQueue（缓存失效、定时任务 ）
  - 是一个支持延时获取元素的无界阻塞队列, 队列使用 PriorityQueue 来实现
  - 队列中的元素必须实现 Delayed 接口，在创建元素时可以指定多久才能从队列中获取当前元素
  - 只有在延迟期满时才能从队列中提取元素
- SynchronousQueue（不存储数据、可用于传递数据）
  - SynchronousQueue是一个不存储元素的阻塞队列，每一个 put 操作必须等待一个 take 操作，否则不能继续添加元素。
  - 它支持公平访问队列。默认情况下线程采用非公平性策略访问队列
  - SynchronousQueue 可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程
  - 队列本身并不存储任何元素，非常适合于传递性场景,比如在一个线程中使用的数据，传递给另外一个线程使用， SynchronousQueue 的吞吐量高于 LinkedBlockingQueue 和 ArrayBlockingQueue
- LinkedTransferQueue
  - LinkedTransferQueue是一个由链表结构组成的无界阻塞TransferQueue队列
  - 相对于其他阻塞队列，LinkedTransferQueue多了tryTransfer和transfer方法
  - transfer方法
    - 如果当前有消费者正在等待接收元素（消费者使用take()方法或带时间限制的poll()方法时），transfer方法可以把生产者传入的元素立刻transfer（传输）给消费者。如果没有消费者在等待接收元素，transfer方法会将元素存放在队列的tail节点，并等到该元素被消费者消费了才返回
  - tryTransfer方法
    - tryTransfer方法是用来试探生产者传入的元素是否能直接传给消费者。如果没有消费者等待接收元素，则返回false。和transfer方法的区别是tryTransfer方法无论消费者是否接收，方法立即返回，而transfer方法是必须等到消费者消费了才返回
- LinkedBlockingDeque

    - LinkedBlockingDeque是一个由链表结构组成的双向阻塞队列，所谓双向队列指的是可以从队列的两端插入和移出元素
    - 双向队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争
    - 相比其他的阻塞队列，LinkedBlockingDeque多了addFirst、addLast、offerFirst、offerLast、peekFirst和peekLast等方法，以First单词结尾的方法，表示插入、获取（peek）或移除双端队列的第一个元素；以Last单词结尾的方法，表示插入、获取或移除双端队列的最后一个元素；另外，插入方法add等同于addLast，移除方法remove等效于removeFirst，但是take方法却等同于takeFirst
- 阻塞队列的实现原理
    - 阻塞队列使用通知模式实现。通知模式，就是当生产者往满的队列里添加元素时会阻塞住生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用

4.24 JUC ForkJoin

- 核心思想: 分治算法(Divide-and-Conquer)
    - 分治算法(Divide-and-Conquer)把任务递归的拆分为各个子任务，最终汇总每个小任务结果后得到大任务结果
- work-stealing(工作窃取)算法
    - 工作窃取算法是指某个线程尝试找到并执行已经提交的任务，或者从其他队列里窃取任务来执行
    - 算法的优点：充分利用线程进行并行计算，减少了线程间的竞争
        - 这种优点使得 ForkJoinPool 在运行多个可以产生子任务的任务，或者是提交的许多小任务时效率更高
        - 尤其是构建异步模型的 ForkJoinPool 时，对不需要合并(join)的事件类型任务也非常适用
    - 算法的缺点：双端队列里只有一个任务时还是存在竞争，并且算法会消耗更多的系统资源，比如创建多个线程和多个双端队列
- Fork/Join框架
    - Fork/Join框架主要包含三个模块
        - **任务对象: ForkJoinTask** (包括RecursiveTask、RecursiveAction 和 CountedCompleter)
        - **线程池: ForkJoinPool**
        - 执行Fork/Join任务的线程: ForkJoinWorkerThread
    - ForkJoinTask 提供在任务中执行fork()和join()操作的机制，通常情况下，只需要继承ForkJoinTask的子类实现需求
        - RecursiveAction：用于没有返回结果的任务
        - RecursiveTask：用于有返回结果的任务
        - CountedCompleter：在任务完成执行后会触发执行一个自定义的钩子函数
    - ForkJoinPool 执行ForkJoinTask
        - ForkJoinPool 只接收 ForkJoinTask 任务，可以通过池中的ForkJoinWorkerThread来处理ForkJoinTask任务
- WorkQueue 任务队列

- 在 ForkJoinPool 中，线程池中每个工作线程(ForkJoinWorkerThread)都对应一个任务队列(WorkQueue)，工作线程优先处理来自自身队列的任务(LIFO或FIFO顺序，参数 mode 决定)，然后以FIFO的顺序随机窃取其他队列中的任务
- 具体思路如下:
  - 每个线程都有自己的一个WorkQueue，该工作队列是一个双端队列
  - 队列支持三个功能push、pop、poll
  - push/pop只能被队列的所有者线程调用，而poll可以被其他线程调用
  - 划分的子任务调用fork时，都会被先push到自己的队列中
  - 如果没有超出最大线程规定数，子任务生成一个新的线程，以及一个新的WorkQueue，
  - 因为WorkQueue是空的，随机从另一个线程的队列末尾调用poll方法窃取任务



- WorkQueue是双端队列，被窃取任务线程从双端队列的头部拿任务执行，而窃取任务的线程从双端队列的尾部拿任务执行

- ForkJoin 例子
  - 采用Fork/Join来异步计算1+2+3+...+10000的结果
  - 定义一个RecursiveTask

```
private class Sumtask extend RecursiveTask<Long>{
    private static final int THRESHOLD = 50; // 阈值
    private int begin;
    private int end;
    public Sumtask(int begin, int end){
        this.begin = begin;
        this.end = end;
    }

    @override
    protected Long computer(){
        long sum = 0;
```

```
        // 如果任务足够小就计算任务
        if (end - begin < THRESHOLD){
            for (int i = begin; i < end; i++)
                sum += i;
        }else{
            // 如果任务大于阈值，就分裂成两个子任务计算
            int mid = (begin + end)/2;
            Sumtask task1 = new Sumtask(begin, mid);
            Sumtask task2 = new Sumtask(mid+1, end);
            // 执行子任务
            task1.fork();
            task2.fork();
            // 等待子任务执行完，并得到其结果
            long sum1 = task1.join();
            long sum2 = task2.join();
            // 合并子任务
            sum = sum1+sum2;
        }
        return sum;
    }

    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        // 生成一个计算任务，负责计算1+2+ ... +9999+10000
        ForkJoinTask<Integer> task = new SumTask(1, 10000);
        // 执行一个任务
        Future<Integer> result = forkJoinPool.submit(task);
        System.out.println(result.get());
    }
}
```

- Fork/Join异常处理
  - ForkJoinTask在执行的时候可能会抛出异常，但是我们没办法在主线程里直接捕获异常，所以ForkJoinTask提供了isCompletedAbnormally()方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过ForkJoinTask的getException方法获取异常

```
if(task.isCompletedAbnormally()){
    System.out.println(task.getException());
}
```

  - getException方法返回Throwable对象，如果任务被取消了则返回CancellationException
  - 如果任务没有完成或者没有抛出异常则返回null

4.25 JUC Atomic CAS

- 什么是CAS
  - CAS的全称为Compare-And-Swap，直译就是对比交换。是一条CPU的原子指令，其作用是让CPU先进行比较两个值是否相等，然后原子地更新某个位置的值，其实现方式是基于硬件平台的交换指令CMPXCHG
  - 所以CAS是靠硬件实现的，JVM只是封装了汇编调用，那些AtomicInteger类便是使用了这些封装后的接口
  - 简单解释：CAS操作包含三个参数，内存位置(V), 预期原值(A), 和一个新值(B)，操作期间先比较当前内存位置V的旧值是否等于预期原值A，如果没有发生变化，交换成新值B，发生了变化则不交换
  - CAS操作是原子性的，所以多线程并发使用CAS更新数据时，可以不使用锁
  - CAS 方式为乐观锁，synchronized 为悲观锁。因此使用 CAS 解决并发问题通常情况下性能更优
- CAS 三大问题
  - **ABA问题**: 因为CAS需要在操作值的时候，检查值有没有发生变化，比如没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时则会发现它的值没有发生变化，但是实际上却变化了
  - ABA问题的解决思路就是使用版本号, 在变量前面追加上版本号, 每次变量更新的时候把版本号加1
  - JDK的Atomic包里提供了一个类AtomicStampedReference, 这个类的compareAndSet方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值
  - **循环时间长开销大**: 自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销
  - **只能保证一个共享变量的原子操作**: 当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁
- Unsafe类
  - Unsafe是位于sun.misc包下的一个类，主要提供一些用于执行低级别、不安全操作的方法，如直接访问系统内存资源、自主管理内存资源等，这些方法在提升Java运行效率、增强Java语言底层资源操作能力方面起到了很大的作用
  - 但由于Unsafe类使Java语言拥有了类似C语言指针一样操作内存空间的能力，这无疑也增加了程序发生相关指针问题的风险

4.26 JUC Atomic 原子操作类

- Java从1.5开始提供了java.util.concurrent.atomic包 (简称Atomic包)，这个包中的原子操作类提供了一种用法简单、性能高效、线程安全地更新一个变量的方式，Atomic包里一共提供了12个类，属于4种类型的原子更新方式，分别是原子更新基本类型、原子更新数组、原子更新引用和原子更新字段
- 原子更新基本类型
  - AtomicBoolean: 原子更新布尔类型
  - AtomicInteger: 原子更新整型 `java.util.concurrent.atomic.AtomicInteger`
  - AtomicLong: 原子更新长整型
- 原子更新数组

    - AtomicIntegerArray: 原子更新整型数组里的元素
    - AtomicLongArray: 原子更新长整型数组里的元素
    - AtomicReferenceArray: 原子更新引用类型数组里的元素
    - 常用两个方法：get(int index)，compareAndSet(int i,E expect,E update)
- 原子更新引用类型
    - AtomicReference: 原子更新引用类型
    - AtomicStampedReference: 原子更新引用类型, 内部使用Pair来存储元素值及其版本号
        - 解决CAS的ABA问题
        - 如果元素值和版本号都没有变化，并且和新的也相同，返回true
        - 如果元素值和版本号都没有变化，并且和新的不完全相同，就构造一个新的Pair对象并执行CAS更新pair
    - AtomicMarkableReferce: 原子更新带有标记位的引用类型
- 原子更新字段类
    - AtomicIntegerFieldUpdater: 原子更新整型的字段的更新器
    - AtomicLongFieldUpdater: 原子更新长整型字段的更新器
    - AtomicReferenceFieldUpdater
        - 字段必须是volatile类型的，在线程之间共享变量时保证立即可见
        - 只能是实例变量，不能是类变量，也就是说不能加static关键字
        - 只能是可修改变量，不能使final变量，因为final的语义就是不可修改
    - 原子地更新字段类需要两步：
        - 因为原子更新字段类都是抽象类，每次使用的时候必须使用静态方法newUpdater()创建一个更新器，并且需要设置想要更新的类和属性
        - 更新类的字段必须使用public volatile修饰

4.27 JUC Tools CountDownLatch

- CountDownLatch允许一个或多个线程等待其他线程完成操作
- 应用场景
    - 需要解析一个Excel里多个sheet的数据，每个线程解析一个sheet里的数据，等到所有的sheet都解析完之后，程序需要提示解析完成。在这个需求中，要实现主线程等待所有线程完成sheet的解析操作

```
public class CountDownLatchTest{
    static CountDownLatch c = new CountDownLatch(2);
    public static void main(String[] args) throws InterruptedException {
        Thread parser = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(1);
                c.countDown();
                System.out.println(2);
```

```
                c.countDown();
            }
            });
        parser.start();
        c.await(); //阻塞当前线程，直至归零
        System.out.println("3");
    }
```

- CountDownLatch表示计数器，构造函数接收一个int类型的参数作为计数器
  - 调用CountDownLatch的await()将会阻塞当前线程，直至归零
  - 其他线程调用CountDownLatch的countDown()方法来对计数器减1，当数字被减成0后，所有await的线程都将被唤醒
  - 对应的底层原理就是，调用await()方法的线程会利用AQS排队，一旦数字被减为0，则会将AQS中排队的线程依次唤醒

```
static CountDownLatch countDownLatch = new CountDownLatch(3);
countDownLatch.await(); // 开始阻塞
countDownLatch.countDown(); // 等三次才会被唤醒
```

- https://pdai.tech/md/java/thread/java-thread-x-juc-tool-countdownlatch.html

4.28 JUC Tools CyclicBarrier

- CyclicBarrier的字面意思是可循环使用（Cyclic）的屏障（Barrier），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运行
- 应用场景
  - CyclicBarrier用于多线程计算数据，最后合并计算结果的场景。例如，用一个Excel保存了用户所有银行流水，每个Sheet保存一个账户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个sheet里的银行流水，都执行完之后，得到每个sheet的日均银行流水，最后，再用barrierAction用这些线程的计算结果，计算出整个Excel的日均银行流水
- CyclicBarrier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

```
public class CyclicBarrierTest {
    staticCyclicBarrier c = new CyclicBarrier(2);
    public static void main(String[] args){
        Thread parser = new Thread(new Runnable(){
            @Override
            public void run(){
                c.await();
                System.out.println(1);
            }
        });
        parser.start();
        c.await();
```

```
    }
}
```

- CountDownLatch和CyclicBarrier
  - 相同在于对应的线程都完成工作之后再进行下一步动作
  - 区别是进行下一步动作的动作实施者不一样
    - CountDownLatch是主线程(即执行main函数)等待其他线程的那个线程就绪后, 它自己run
    - CyclicBarrier是执行任务的线程大伙儿，一起同时做某事
  - CountDownLatch的计数器只能使用一次，而CyclicBarrier的计数器可以使用reset()方法重置
- https://pdai.tech/md/java/thread/java-thread-x-juc-tool-cyclicbarrier.html

4.29 JUC Tools Semaphore

- Semaphore (信号量) 是用来控制同时访问特定资源的线程数量，它通过协调各个线程，以保证合理的使用公共资源
- 应用场景
  - Semaphore可以用于做流量控制，特别是公用资源有限的应用场景，比如数据库连接
  - 需求是要读取几万个文件的数据，因为都是IO密集型任务，我们可以启动几十个线程并发地读取，但是如果读到内存后，还需要存储到数据库中，而数据库的连接数只有10个，这时我们必须控制只有10个线程同时获取数据库连接保存数据，否则会报错无法获取数据库连接，可以使用Semaphore来做流量控制

```
public class SemaphoreTest {
    private static final int THREAD_COUNT = 30;
    private static ExecutorServicethreadPool =
Executors.newFixedThreadPool(THREAD_COUNT);
    private static Semaphore s = new Semaphore(10);
    public static void main(String[] args) {
        for (inti = 0; i< THREAD_COUNT; i++) {
            threadPool.execute(new Runnable() {
                @Override
                public void run(){
                    s.acquire();
                    System.out.println("save data");
                    s.release();
                }
            });
            threadPool.shutdown();
        }
    }
}
```

  - 虽然有30个线程在执行，但是只允许10个并发执行
- 构造方法Semaphore（int permits）接受一个整型的数字，表示同时可用的许可证数量
  - 线程通过acquire()来获取许可，如果没有许可可用则线程阻塞，并通过AQS来排队；通过release()方法来释放许可

- 当某个线程释放了某个许可后，会从AQS中正在排队的第一个线程开始依次唤醒，直到没有空闲许可

```
static Semaphore semaphore = new Semaphore(3);
semaphore.acquire(); // 可以有三个获取许可
semaphore.release();
```

- https://pdai.tech/md/java/thread/java-thread-x-juc-tool-semaphore.html

4.30 JUC Tools Exchanger

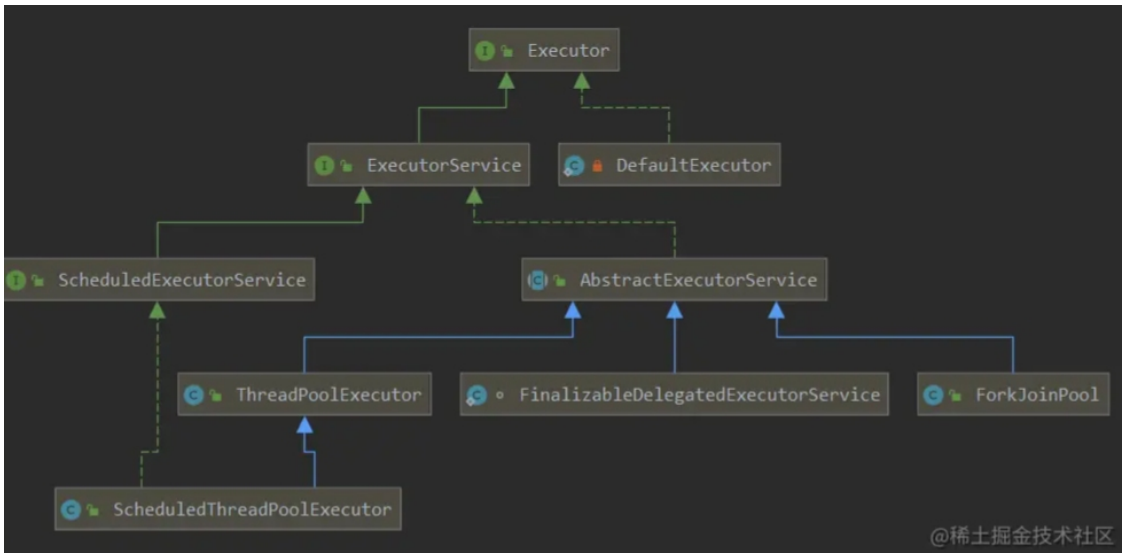
- Exchanger (交换者) 是一个用于线程间协作的工具类，用于进行线程间的数据交换
- 它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。两个线程通过exchange方法交换数据，如果第一个线程先执行exchange()方法，它会一直等待第二个线程也执行exchange方法，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方
- 应用场景
  - Exchanger可以用于遗传算法，遗传算法里需要选出两个人作为交配对象，这时候会交换两人的数据，并使用交叉规则得出2个交配结果
  - Exchanger也可以用于校对工作，需要将纸制银行流水通过人工的方式录入成电子银行流水，为了避免错误，采用AB岗两人进行录入，录入到Excel之后，系统需要加载这两个Excel，并对两个Excel数据进行校对，看看是否录入一致

```
public class ExchangerTest {
    private static final Exchanger<String>exgr = new Exchanger<String>();
    private static ExecutorServicethreadPool =
Executors.newFixedThreadPool(2);
    public static void main(String[] args) {
        threadPool.execute(new Runnable() {
            @Override
            public void run() {
                String A = "银行流水A";
                exgr.exchange(A);
            }
        });
        threadPool.execute(new Runnable() {
            @Override
            public void run() {
                String B = "银行流水B";
                String A = exgr.exchange("B");
                System.out.println(A.equals(B) + "，A录入的是："+ A + "，B录入
是：" + B);
            }
        });
        threadPool.shutdown();
    }
}
```

- 如果两个线程有一个没有执行exchange()方法，则会一直等待，如果担心有特殊情况发生，避免一直等待，可以使用exchange（V x，longtimeout，TimeUnit unit）设置最大等待时长
- https://pdai.tech/md/java/thread/java-thread-x-juc-tool-exchanger.html

4.31 JUC 线程池 Executor

- 几乎所有需要异步或并发执行任务的程序都可以使用线程池
  - 降低资源消耗 (重复利用已创建的线程降低线程创建和销毁造成的消耗)
  - 提高响应速度(任务能立即执行无须创建线程)
  - 提高线程的可管理性 (线程池统一分配、调优和监控)
- Executor
  - 应用程序通过**Executor框架**控制上层的调度，操作系统内核**OSKernel**控制下层的调度
- Executor框架主要由3大部分组成
  - 任务：包括被执行任务需要实现的接口：Runnable接口或Callable接口
  - 任务的执行：包括任务执行机制的核心接口Executor，以及继承自Executor的ExecutorService接口
  - 异步计算的结果：包括接口Future和实现Future接口的FutureTask类
- Executor管理多个异步任务的执行，无需程序员显式地管理线程的生命周期手动创建线程池
  - 异步是指多个任务的执行互不干扰，不需要进行同步操作
  - `Executor` 是顶层接口，`Executor` 中只有一个 `execute` 方法，用于执行任务。线程的创建、调度等细节均由其子类实现



  - ThreadPoolExecutor是Executor的核心实现类，用来执行被提交的任务
  - ScheduledThreadPoolExecutor是ThreadPoolExecutor的实现类，可以在给定的延迟后运行命令，或者定期执行命令
  - Future接口和实现Future接口的FutureTask类，代表异步计算的结果
  - Runnable接口和Callable接口的实现类，都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行
- 配置线程池需要考虑因素

- 任务的优先级，任务的执行时间长短，任务的性质(CPU密集/ IO密集)，任务的依赖关系
- 性质不同的任务可用使用不同规模的线程池分开处理
  - CPU密集型: 尽可能少的线程，Ncpu+1
  - IO密集型: 尽可能多的线程, Ncpu*2，大部分时间阻塞在I/O上, 比如数据库连接池

4.32 JUC 线程池 ThreadPoolExecutor

- 线程池实现原理
  - 线程池判断当前运行的线程少于corePoolSize
    - 如果核心线程池里的线程没满，创建一个新线程 (需要获取全局锁)
    - 如果核心线程池里的线程等于或多于corePoolSize，则将任务加入BlockingQueue
  - 线程池判断BlockingQueue是否已经满
    - 如果BlockingQueue没有满，则将新提交的任务存储在这个BlockingQueue里
    - 如果无法将任务加入BlockingQueue（队列已满），则创建新的线程来处理任务
  - 线程池判断当前运行的线程少于maximumPoolSize
    - 如果没有，则创建一个新的工作线程来执行任务 (需要获取全局锁)
    - 如果当前运行的线程超出maximumPoolSize，执行饱和策略来处理这个任务

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    // 如果线程数小于基本线程数，则创建线程并执行当前任务
    if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {
        // 如线程数大于等于基本线程数或线程创建失败，则将当前任务放到工作队列中。
        if (runState == RUNNING && workQueue.offer(command)) {
            if (runState != RUNNING || poolSize == 0)
                ensureQueuedTaskHandled(command);
        }
        // 如果线程池不处于运行中或任务无法放入队列，并且当前线程数量小于最大允许的线程数
量，
        // 则创建一个线程执行任务。
        else if (!addIfUnderMaximumPoolSize(command))
            // 抛出RejectedExecutionException异常
        reject(command); // is shutdown or saturated
    }
}
```

  - 为什么当核心线程数已满时，是先添加队列，而不是先创建临时线程
    - 在创建新线程的时候，需要获取全局锁，这个时候其他线程就得被阻塞，影响了线程的整体执行效率。而将任务添加到队列缓冲，很好的避免了临时线程的创建销毁开销
- 线程池的创建
  - 通过ThreadPoolExecutor来创建一个线程池

```
new ThreadPoolExecutor(corePoolSize,
                       maximumPoolSize,
                       keepAliveTime,
                       milliseconds,
                       runnableTaskQueue, handler);
```

    - 创建一个线程池时需要输入几个参数
        - corePoolSize（线程池的基本大小）
        - runnableTaskQueue（任务队列）用于保存等待执行的任务的阻塞队列
        - maximumPoolSize（线程池最大数量）
        - ThreadFactory：用于设置创建线程的工厂
        - RejectedExecutionHandler（饱和策略）
- 线程池提交任务
    - 可以使用两个方法向线程池提交任务，分别为execute()和submit()方法
    - execute()方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功

      `threadsPool.execute(new Runnable(){@Override public void run() {} });`
    - submit()方法用于提交需要返回值的任务。线程池会返回一个future类型的对象，可以判断任务是否执行成功，通过future的get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成

      `Future<Object> future = executor.submit(harReturnValuetask);`
- 关闭线程池
    - 可以通过调用线程池的`shutdown`或`shutdownNow`方法来关闭线程池
    - 原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程
    - shutdownNow首先将线程池的状态设置成STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表
    - shutdown只是将线程池的状态设置成SHUTDOWN状态，然后中断所有没有正在执行任务的线程
    - 要调用了这两个关闭方法中的任意一个，isShutdown方法就会返回true
    - 当所有的任务都已关闭后，才表示线程池关闭成功，这时调用isTerminaed方法会返回true
- ThreadPoolExecutor继承自AbstractExecutorService，通常由工厂类Executors来创建
- 主要有三种 ThreadPoolExecutor:
    - FixedThreadPool: 所有任务只能使用固定大小的线程
        - FixedThreadPool的工作队列为无界队列LinkedBlockingQueue(队列容量为Integer.MAX_VALUE)

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

        - 适用于为了满足资源管理的需求，而需要限制当前线程数量的应用场景，它适用于负载比较重的服务器

```
ExecutorService executor = Executors.newFixedThreadPool(5);
```

    - SingleThreadExecutor: 相当于大小为 1 的 FixedThreadPool
        - 线程池中只有一个线程，如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一线程可以保证任务顺序执行

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

        - 适用于需要保证顺序地执行各个任务；并且在任意时间点，不会有多个线程是活动的应用场景

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

    - CachedThreadPool: 一个任务创建一个线程
        - 适用于执行很多的短期异步任务的小程序，或者是负载较轻的服务器

```
public static ExecutorService newCachedThreadPool() {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>
());
}
```

        - 线程池的线程数可达到Integer.MAX_VALUE，即2147483647，内部使用没有容量的SynchronousQueue作为阻塞队列，这意味着，如果主线程提交任务的速度高于maximumPool中线程处理任务的速度时，CachedThreadPool会不断创建新线程
        - 和newFixedThreadPool创建的线程池不同，newCachedThreadPool在没有任务执行时，当线程的空闲时间超过keepAliveTime，会自动释放线程资源, 当提交新任务时，如果没有空闲线程，则创建新线程执行任务

```
ExecutorService executor = Executors.newCachedThreadPool();
```

- 例子：创建了固定大小为五个工作线程的线程池, 然后分配给线程池十个工作

```
public class SimpleThreadPool {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread("" + i);
```

```
            executor.execute(worker);
        }
        executor.shutdown();
        while (!executor.isTerminated()) {}
        System.out.println("Finished all threads");
    }
}
public class WorkerThread implements Runnable {
    private String command;
    public WorkerThread(String s){
        this.command=s;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()+" Start. Command =
"+command);
        System.out.println(Thread.currentThread().getName()+" End.");
    }

    @Override
    public String toString(){
        return this.command;
    }
}
```

4.33 JUC 线程池 ScheduledThreadPoolExecutor

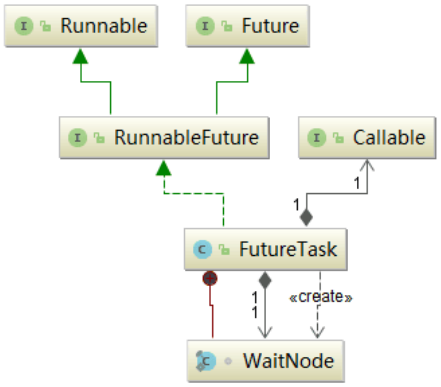
- 在很多业务场景中，我们可能需要周期性的运行某项任务来获取结果，比如周期数据统计，定时发送数据等
- ScheduledThreadPoolExecutor简介
  - ScheduledThreadPoolExecutor继承自 ThreadPoolExecutor，为任务提供延迟或周期执行
  - 使用专门的任务类型 ScheduledFutureTask 来执行周期任务，也可以接收不需要时间调度的任务
  - 使用专门的存储队列 DelayedWorkQueue 来存储任务，DelayedWorkQueue 是无界延迟队列 DelayQueue 的一种
  - 支持可选的run-after-shutdown参数，在池被关闭 (shutdown) 之后支持可选的逻辑来决定是否继续运行周期或延迟任务。并且当任务(重新)提交操作与 shutdown 操作重叠时，复查逻辑也不相同
  - ScheduledThreadPoolExecutor的执行主要分为两大部分
    - 调用ScheduledThreadPoolExecutor的scheduleAtFixedRate()方法或者scheduleWithFixedDelay()方法，会向ScheduledThreadPoolExecutor的DelayQueue添加一个实现了RunnableScheduledFutur接口的ScheduledFutureTask
    - 线程池中的线程从DelayQueue中获取ScheduledFutureTask，然后执行任务
- 数据结构

    - ScheduledThreadPoolExecutor继承了ThreadPoolExecutor，ScheduledThreadPoolExecutor还实现了ScheduledExecutorService接口，这个接口规定了一些方法签名，这些方法负责把周期性任务提交到线程池
    - ScheduledThreadPoolExecutor 内部构造了两个内部类 ScheduledFutureTask 和 DelayedWorkQueue
    - ScheduledFutureTask:
        - 继承了FutureTask，说明是一个异步运算任务，并且最上层分别实现了Runnable、Future、Delayed接口，说明它是一个可以延迟执行的异步运算任务
        - ScheduledFutureTask主要包含3个成员变量：long型 time，long型 sequenceNumber，long型 period
    - DelayedWorkQueue:
        - 这是 ScheduledThreadPoolExecutor 为存储周期或延迟任务专门定义的一个延迟队列，继承了 AbstractQueue，为了契合 ThreadPoolExecutor 也实现了 BlockingQueue 接口
        - 它内部只允许存储 RunnableScheduledFuture 类型的任务。与 DelayQueue 的不同之处就是它只允许存放 RunnableScheduledFuture 对象，并且自己实现了二叉堆(DelayQueue 是利用了 PriorityQueue 的二叉堆结构)
        - DelayQueue封装了一个PriorityQueue，这个PriorityQueue会对队列中的ScheduledFutureTask进行排序，time小的排在前面（时间早的任务将被先执行）。如果两个ScheduledFutureTask的time相同，就比较sequenceNumber，sequenceNumber小的排在前面（先提交的任务将被先执行）
- 计划任务分为两种
    - 非周期性任务，这种任务只执行一次，需要在指定的时间运行
    - 周期性任务，这种任务要执行多次，周期性任务又可以分为两种
        - 固定频率：每隔一段时间，任务就执行一次，比如每五分钟执行一次
        - 固定间隔：两次任务的执行之间需要间隔一定的时间，比如本次任务执行后，等待五分钟，然后执行下一次任务
- ScheduledThreadPoolExecutor相比ThreadPoolExecutor有哪些特性?
    - 使用DelayQueue作为任务队列
    - **多次执行任务**：工作线程`worker`在工作时，会从工作队列`workQueue`中提取任务，然后执行任务，本次任务执行完以后，设定任务下一次执行的时候，然后将任务再次放入工作队列`workQueue`，工作线程`worker`就可以再次从工作队列`workQueue`中提取这个任务，然后执行，周而复始，就可以做到多次执行任务
    - **在指定时间执行任务**：`ScheduledThreadPoolExecutor`使用特定的工作队列`DelayedWorkQueue`实现，工作线程`worker`在工作时，会从工作队列`workQueue`中提取任务，在提取任务时，如果任务还没有到执行的时间，那么工作线程`worker`就会阻塞一段时间，直到任务的执行时间到来，工作线程`worker`自动唤醒，成功从工作队列`workQueue`中提取任务，然后执行
    - 通过阻塞的方式，让工作线程`worker`进入阻塞，直到任务执行时间到来，工作线程才能成功拿到任务并执行，这就可以做到：任务只有在指定时间到来以后，才能执行
- 提交任务的四个方法

    - schedule(无返回值)
    - schedule(有返回值)
    - scheduledAtFixedRate
        - 多次执行任务，创建任务后，经过 initialDelay 时间，执行第一次任务
        - 此后，每隔 period 时间，执行一次任务，无论上一次任务是否完成，都会执行
    - scheduledAtFixedDelay
        - 多次执行任务，创建任务后，经过 initialDelay 时间，执行第一次任务
        - 每次任务执行完成之后，间隔 delay 时间，才执行下一次任务
- ScheduledThreadPoolExecutor 三部分内容
    - 任务提交：ScheduledThreadPoolExecutor.schedule()方法
    - 任务执行：ScheduledThreadPoolExecutor.ScheduledFutureTask.run()方法
    - 工作队列：ScheduledThreadPoolExecutor.DelayedWorkQueue类中的take()方法和offer()方法
- Executors 提供了几种方法来构造 ScheduledThreadPoolExecutor?
    - newScheduledThreadPool: 可指定核心线程数的线程池
    - newSingleThreadScheduledExecutor: 只有一个工作线程的线程池。如果内部工作线程由于执行周期任务异常而被终止，则会新建一个线程替代它的位置
- 计划任务线程池ScheduledThreadPoolExecutor原理

4.34 JUC 线程池 FutureTask

- FutureTask简介
    - Future 表示了一个任务的生命周期，是一个可取消的异步运算，可以把它看作是一个异步操作的结果的占位符，它将在未来的某个时刻完成，并提供对其结果的访问。在并发包中许多异步任务类都继承自Future，其中最典型的就是 FutureTask
    - FutureTask 为 Future 提供了基础实现，如获取任务执行结果(get)和取消任务(cancel)等。如果任务尚未完成，获取任务执行结果时将会阻塞。一旦执行结束，任务就不能被重启或取消(除非使用runAndReset执行计算)。
    - FutureTask实现了RunnableFuture接口，则RunnableFuture接口继承了Runnable接口和Future接口，所以FutureTask既能当做一个Runnable直接被Thread执行，也能作为Future用来得到Callable的计算结果



- FutureTask源码
  - Callable接口
    - Callable是个泛型接口，泛型V就是要call()方法返回的类型
  - Future接口
    - Future接口代表异步计算的结果，通过Future接口提供的方法可以查看异步计算是否执行完成，或者等待执行结果并获取执行结果，同时还可以取消执行。
    - ```
      public interface Future<V> {
          boolean cancel(boolean mayInterruptIfRunning);
          boolean isCancelled();
          boolean isDone();
          V get() throws InterruptedException, ExecutionException;
          V get(long timeout, TimeUnit unit)
              throws InterruptedException, ExecutionException,
      TimeoutException;
      }
      ```
    - FutureTask.cancel()方法用来取消异步任务的执行。如果异步任务已经完成或者已经被取消，或者由于某些原因不能取消，则会返回false。如果任务还没有被执行，则会返回true并且异步任务不会被执行。如果任务已经开始执行了但是还没有执行完成，若mayInterruptIfRunning为true，则会立即中断执行任务的线程并返回true，若mayInterruptIfRunning为false，则会返回true且不会中断任务执行线程
    - FutureTask.isCanceled() 判断任务是否被取消，如果任务在结束(正常执行结束或者执行异常结束)前被取消则返回true，否则返回false
    - FutureTask.isDone() 判断任务是否已经完成，如果完成则返回true，否则返回false。需要注意的是：任务执行过程中发生异常、任务被取消也属于任务已完成，也会返回true
    - FutureTask.get() 获取任务执行结果，如果任务还没完成则会阻塞等待直到任务执行完成。如果任务被取消则会抛出CancellationException异常，如果任务执行过程发生异常则会抛出ExecutionException异常，如果阻塞等待过程中被中断则会抛出InterruptedException异常
- 构造函数
  - FutureTask(Callable callable)
    - 把传入的Callable变量保存在this.callable字段中
      ```

    - FutureTask(Runnable runnable, V result)
        - 把传入的Runnable封装成一个Callable对象保存在callable字段中, 同时如果任务执行成功的话就会返回传入的result
- 示例代码

```
Callable<String> task = new Callable<String>() {
    public String call() throws InterruptedException {
        return taskName;
    }
};
FutureTask<String> futureTask = new FutureTask<String>(task);
futureTask.run();
return future.get();
```

5. ♥ Java JVM ♥

5.1 基本概念

- JVM 是可运行 Java 代码的假想计算机，包括一套字节码指令集、一个垃圾回收
    - 一个堆、一个存储方法区
    - 一个栈、一组寄存器、本地方法栈
- JVM 是运行在操作系统之上的，它与硬件没有直接的交互
- Java 源文件通过编译器，能够生产相应的.Class 文件，也就是字节码文件; 而字节码文件又通过 Java 虚拟机中的解释器，编译成特定机器上的机器码
- Java程序的执行过程
    - Java同时使用 compiler 和 interpreter
    - **Java编译器** 将 `.java` 源文件 编译成 `.class` 字节码文件
    - **JVM中的Java解释器** 将字节码文件解释成具体硬件环境和操作系统平台下的机器码
    - .java file -> `compiler` -> .class bytecode file -> JVM -> classloader -> `interpreter` -> Machine Code -> 程序运行
- 每一种平台的解释器是不同的，但是实现的虚拟机是相同的，这也就是 Java 为什么能够跨平台的原因

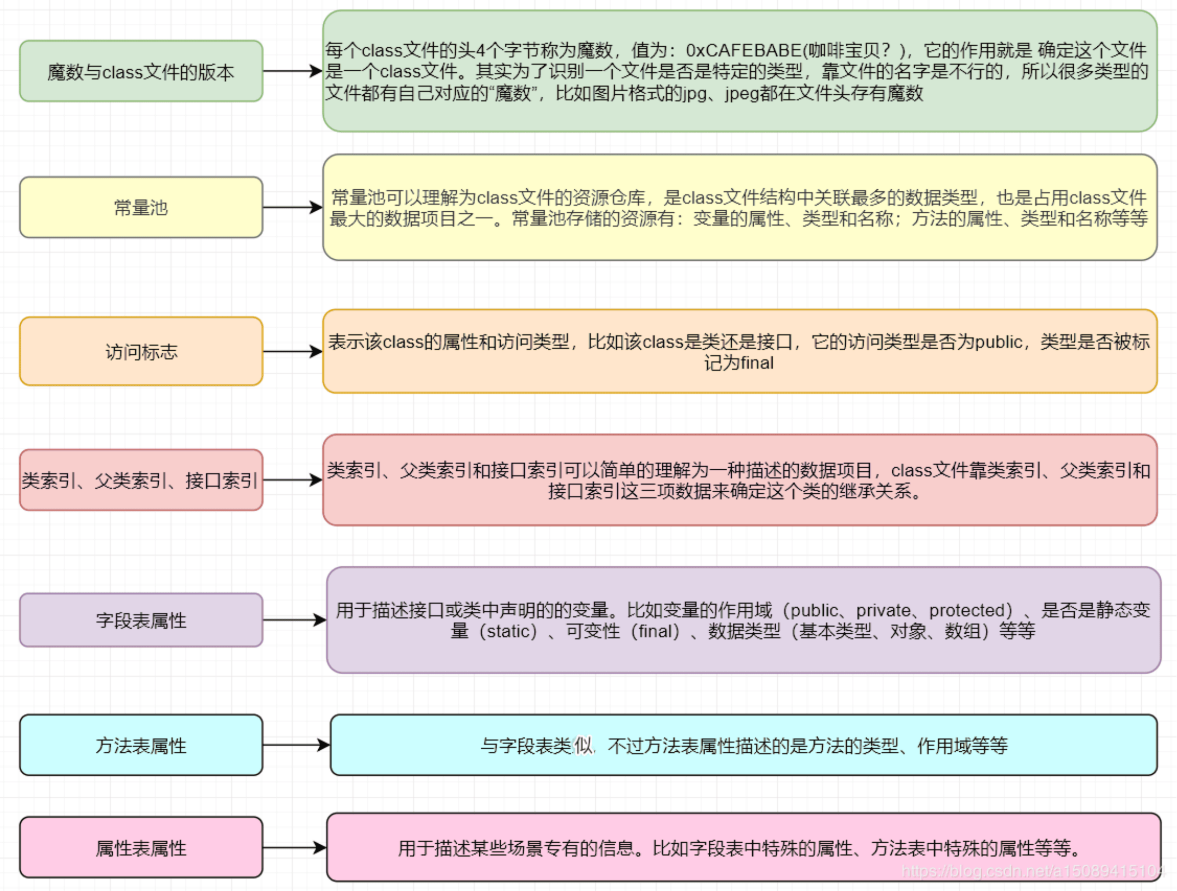
5.2 什么是字节码

- Java是高级语言，计算机无法识别，必须先编译成字节码文件 (.class), 再由java虚拟机运行解释编译后的java代码
    - 编译后的java代码，就是java字节码
- 字节码不面向任何特定的处理器，只面向虚拟机
- 采用字节码的好处
    - Java语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点

    - 由于字节码并不专对一种特定的机器，Java程序无须重新编译便可在多种不同的计算机上运行
    - Java代码翻译成字节码，储存字节码的文件再交由运行于不同平台上的JVM虚拟机去读取执行，实现一次编写，到处运行

- Java字节码文件
    - class文件本质上是一个以8位字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑的排列在class文件中
    - jvm根据其特定的规则解析该二进制数据，从而得到相关信息
    - class文件采用一种伪结构来存储数据，它有两种类型：无符号数和表

- Class文件的结构属性



- 常量池
    - Constant pool意为常量池, 可以理解成Class文件中的资源仓库
    - 主要存放的是两大类常量：字面量(Literal)和符号引用(Symbolic References)
    - 字面量类似于java中的常量概念，如文本字符串，final常量等，而符号引用则属于编译原理方面的概念:
        - 类和接口的全限定名(Fully Qualified Name), 字段的名称和描述符号(Descriptor), 方法的名称和描述符

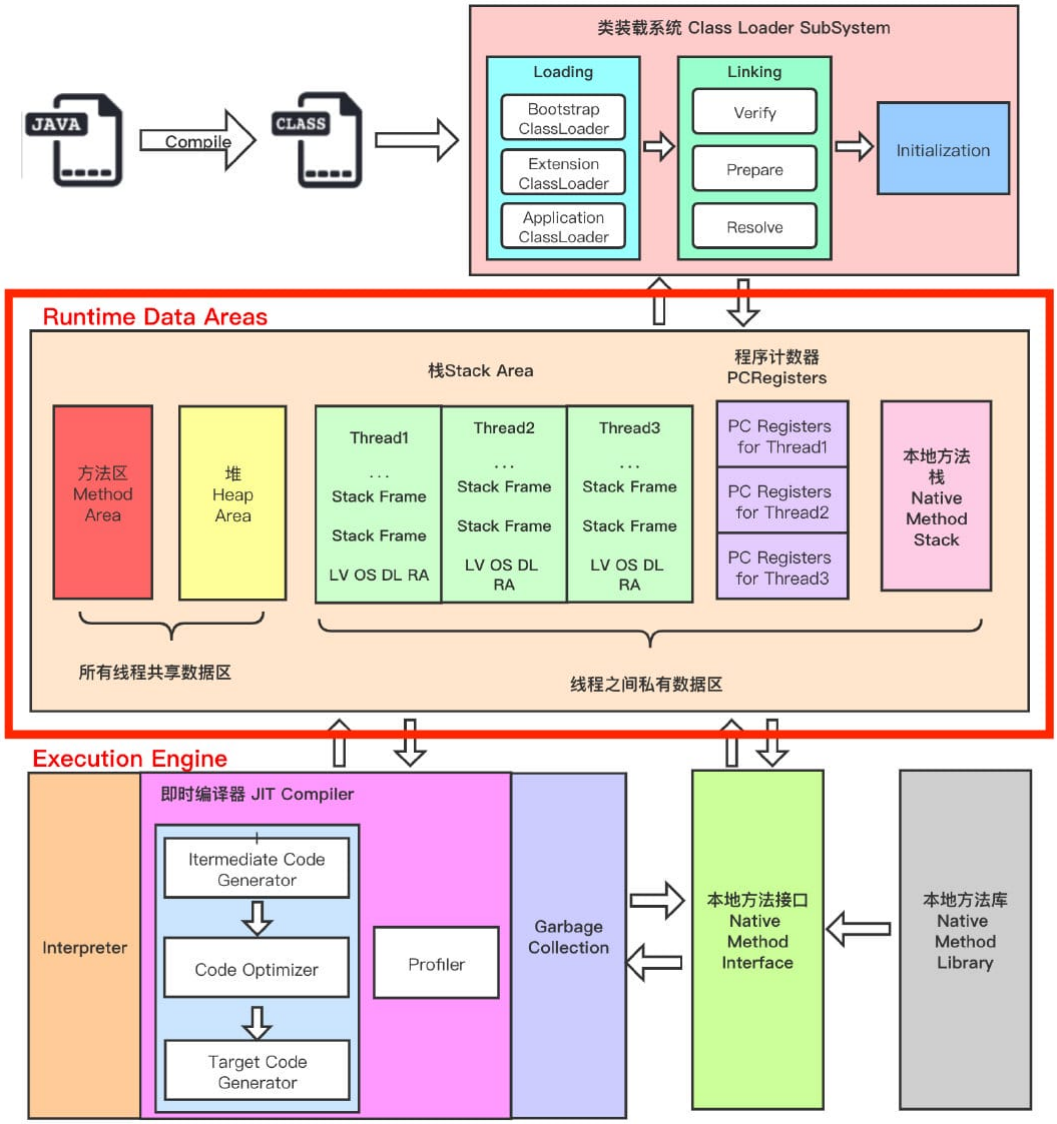
5.3 线程

- JVM 允许一个应用并发执行多个线程
- Java的多线程是基于操作系统的多线程实现的, Java 线程与原生操作系统线程有直接的映射关系
- 当线程本地存储、缓冲区分配、同步对象、栈、程序计数器等准备好以后，就会创建一个操作系统原生线程。Java 线程结束，原生线程随之被回收。操作系统负责调度所有线程，并把它们分配到任何可用的 CPU 上。当原生线程初始化完毕，就会调用 Java 线程的 run() 方法。当线程结束时，会释放原生线程和 Java 线程的所有资源。
- Java在windows和linux上实现线程的方式是内核线程
    - 内核线程（Kernel-Level Thread KLT）就是直接由操作系统内核支持的线程
    - 但是程序一般不会直接使用内核线程。而是去使用内核线程的一种高级接口——轻量级进程（Light Weight Process，LWP）
    - 由于每个轻量级进程都由一个内核线程支持，因此轻量级进程与内核线程之间的关系为1:1，也叫一对一的线程模型
- Java线程的调度方式是抢占式调度
    - 每个线程由系统来分配执行时间，线程的切换不由线程本身决定
    - 在这种模式下，我们有时候会希望给某些线程多一点执行时间。所以JDK引入了线程优先级的概念
- Hotspot JVM 后台运行的系统线程主要有下面几个
    - 虚拟机线程（VM thread）
    - 周期性任务线程：这线程负责定时器事件（也就是中断），用来调度周期性操作的执行
    - GC 线程：这些线程支持 JVM 中不同的垃圾回收活动
    - 编译器线程：这些线程在运行时将字节码动态编译成本地平台相关的机器码
    - 信号分发线程：这个线程接收发送到 JVM 的信号并调用适当的 JVM 方法处理

5.4 JVM 内存结构

- Java内存模型 JMM 是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了Java程序在各种平台下对内存的访问都能保证效果一致的机制及规范。JVM 内存布局规定了 Java 在运行过程中内存申请、分配、管理的策略，保证了 JVM 的高效稳定运行
- JVM 定义了若干种程序运行期间会使用到的运行时数据区，
    - **线程共享数据区域**随着虚拟机启动而创建，随着虚拟机关闭而销毁
    - **线程私有数据区域**则是与线程对应的，这些与线程对应的数据区域会随着线程开始和结束而创建和销毁
- **线程私有 Thread Local：**
    - **程序计数器 PC**: 指向虚拟机字节码指令的位置, 唯一一个无OOM的区域
    - **虚拟机栈 VM Stack**:
        - 虚拟机栈和线程的生命周期相同
        - 一个线程中，每调用一个方法创建一个栈顺 (Stack Frame)
        - 栈帧的结构: 本地变量表 Local Variable, 操作数栈 Operand stack, 运行时常量池的引用 Runtime Constant Pool Reference

        - 异常: 线程请求栈深度大于JVM所允许的深度StackOverflowError, OutofMemoryError
    - **本地方法栈Native Method Stack**:
        - 异常: 线程请求栈深度大于JVM所允许的深度StackOverflowError, OutofMemoryError
- **线程共享 Thread Shared：**
    - **堆 Java Heap**
        - 新生代(eden, from survivor, to survivor), 老生代, 异常OutofMemoryError
    - **方法区 (永久代) Method Area**
        - 运行时常量池 Runtime Constant Pool
- **堆外内存 (off-heap memory) 又叫直接内存(Direct Memory)**
    - 不受JVM GC管理



5.5 JVM 内存结构 程序计数器

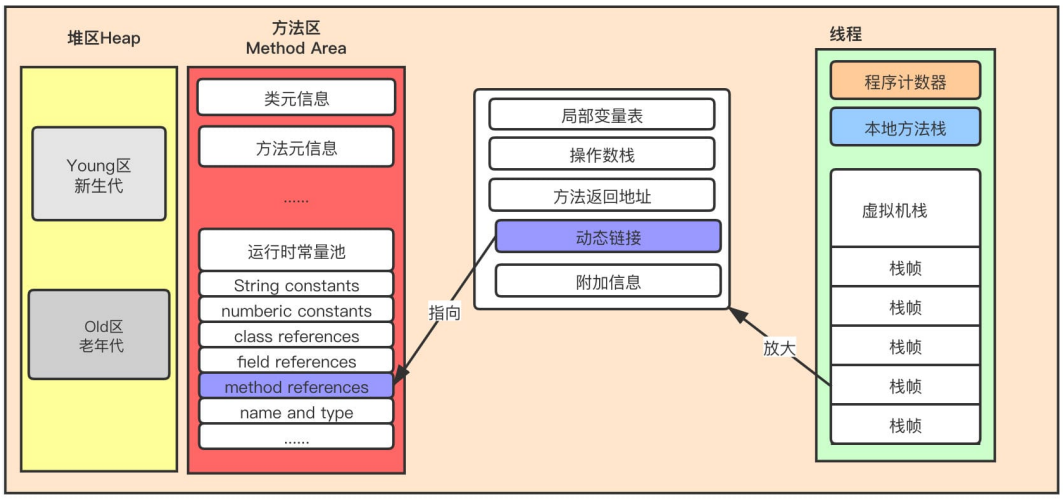
- 程序计数寄存器 (Program Counter Register) 的命名源于 CPU 的寄存器，寄存器存储指令相关的线程信息，CPU 只有把数据装载到寄存器才能够运行。这里寄存器并非是广义上所指的物理寄存器，而叫程序计数器（或PC计数器或指令计数器）
- **JVM 中的 PC 寄存器是对物理 PC 寄存器的一种抽象模拟**
- 程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的**行号指示器**
- 程序计数器作用
  - PC 寄存器用来存储**指向下一条指令的地址**，即将要执行的方法区的字节码，由执行引擎读取下一条指令
  - 由于JVM的多线程是通过线程切换实现的，因为切换的时候要保存目前的线程状态，当切换回当前线程是要还原运行状态，PC就时记录运行到指令的地址
  - JVM的字节码解释器就需要通过改变PC寄存器的值来明确下一条应该执行什么样的字节码指令
- PC寄存器为什么会被设定为**线程私有**的
  - CPU会不停的做任务切换，这样必然会导致经常中断或恢复。为了能够准确的记录各个线程正在执行的当前字节码指令地址，所以为每个线程都分配了一个PC寄存器，每个线程都独立计算，不会互相影响
- 这个内存区域是唯一一个在虚拟机中没有规定任何 OutOfMemoryError 情况的区域

5.6 JVM 内存结构 虚拟机栈

- Java 虚拟机栈(JVM Stacks)描述的是java方法执行的线程内存模型
  - 每个线程在创建的时候都会创建一个虚拟机栈，其内部保存一个个的栈帧(Stack Frame) 对应着一次次 Java 方法调用
  - 每个方法在执行的同时都会创建一个栈帧, 用于存储局部变量表，操作数栈，动态链接以及方法出口等信息
  - 每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程
  - 因为栈帧在运行完成后会弹出，不需要考虑垃圾回收的问题
  - 虚拟机栈是线程私有的，**生命周期和线程一致**
- 作用
  - 主管 Java 程序的运行，它保存方法的局部变量、部分结果，并参与方法的调用和返回
- 特点
  - 栈是一种快速有效的分配存储方式，访问速度仅次于程序计数器
  - JVM 直接对虚拟机栈的操作只有两个：每个方法执行栈帧入栈，方法执行结束栈帧出栈
  - 栈帧随着方法调用而创建，随着方法结束而销毁, 不存在垃圾回收问题
- 栈的存储单位
  - 每个线程都有自己的栈，栈中的数据都是以**栈帧**的格式存在
  - 在这个线程上正在执行的每个方法都各自有对应的一个栈帧
  - 栈帧是一个内存区块，是一个数据集，维系着方法执行过程中的各种数据信息

- 栈运行原理
  - JVM 直接对 Java 栈的操作只有两个，对栈帧的**压栈**和**出栈**，遵循“先进后出/后进先出”原则
  - 在一条活动线程中，一个时间点上，只会有一个活动的栈帧。即只有当前正在执行的方法的栈帧（**栈顶栈帧**）是有效的，这个栈帧被称为**当前栈帧**（Current Frame），与当前栈帧对应的方法就是**当前方法**（Current Method），定义这个方法的类就是**当前类**（Current Class）
- 栈帧的内部结构
  - 每个栈帧中存储着
    - 局部变量表(Local Variables), 操作数栈(Operand Stack), 动态链接(Dynamic Linking), 方法返回地址(Return Address)

- 局部变量表
  - 局部变量表也被称为局部变量数组或者本地变量表, 最基本的存储单元是 Slot（变量槽）
  - 局部变量表是一个数组，被分成很多个slot，主要存放方法参数和局部变量，也会存放reference类型
  - 32 位以内的类型只占用一个 Slot(包括returnAddress类型)，64 位的类型（long和double）占用两个连续的 Slot
    - byte、short、char 在存储前被转换为int，boolean也被转换为int，0 表示 false，非 0 表示 true
    - long 和 double 则占据两个 Slot
  - slot还可以重复使用，如果过了某个引用的作用范围，那么他的位置就可以被重复使用。
  - 局部变量表是重要的垃圾回收根节点集合，只要是局部变量表里直接或间接引用的对象都不会被回收
- 操作数栈
  - 每个独立的栈帧中除了包含局部变量表之外，还包含一个 Last-In-First-Out 的操作数栈，也称为 Expression Stack
  - **操作数栈，在方法执行过程中，根据字节码指令，往操作数栈中写入数据或提取数据，即push pop**
  - 某些字节码指令将值压入操作数栈，其余的字节码指令将操作数取出栈。使用它们后再把结果压入栈
  - 操作数栈的任何一个元素都可以是任意的 Java 数据类型, 32位数据类型占用一个栈单位深度，64位的占用两个
  - **如果被调用的方法带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中**，并更新 PC 寄存器中下一条需要执行的字节码指令
- 动态链接（指向运行时常量池的方法引用）
  - 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，即每个栈帧都有一个指向自己对方法的引用，这个方法信息在运行时常量池中。持有这个引用是为了支持方法调用过程中的动态连接(Dynamic Linking)



  - 在 Java 源文件被编译到字节码文件中时，所有的变量和方法引用都作为**符号引用**（Symbolic Reference）保存在 Class 文件的常量池中。比如：描述一个方法调用了另外的其他方法时，就是通过常量池中指向方法的符号引用来表示的，那么**动态链接的作用就是为了将这些符号引用转换为调用方法的直接引用**
  - JVM 是如何执行方法调用的
    - 方法调用不同于方法执行，方法调用阶段的唯一任务就是确定被调用方法的版本（即调用哪一个方法），暂时还不涉及方法内部的具体运行过程。Class 文件的编译过程中不包括传统编译器中的连接步骤，一切方法调用在 Class文件里面存储的都是符号引用，而不是方法在实际运行时内存布局中的入口地址（直接引用）。也就是需要在类加载阶段，甚至到运行期才能确定目标方法的直接引用
  - 将符号引用转换为调用方法的直接引用与方法的绑定机制有关
    - 静态链接：当一个字节码文件被装载进 JVM 内部时，如果被调用的目标方法在编译期可知，且运行期保持不变时。这种情况下将调用方法的符号引用转换为直接引用的过程称之为静态链接
    - 动态链接：如果被调用的方法在编译期无法被确定下来，也就是说，只能在程序运行期将调用方法的符号引用转换为直接引用，由于这种引用转换过程具备动态性，因此也就被称之为动态链接
  - 对应的方法的绑定机制为：早期绑定（Early Binding）和晚期绑定（Late Binding）
    - 绑定是一个字段、方法或者类在符号引用被替换为直接引用的过程，这仅仅发生一次
    - 被调用的目标方法如果在编译期可知，且运行期保持不变时，即可将这个方法与所属的类型进行绑定, 因此是早期绑定（对应的字节码指令是invokespcial）
    - 被调用的方法在编译器无法被确定下来，只能够在程序运行期根据实际的类型绑定相关的方法，这种绑定方式就被称为晚期绑定（对应的字节码指令是invokevirtual和invokeinterface）
  - 也正是Java的多态特性，才有了早期绑定和晚期绑定。
- 返回地址 (return address)
  - 用来存放调用该方法的 PC 寄存器的值

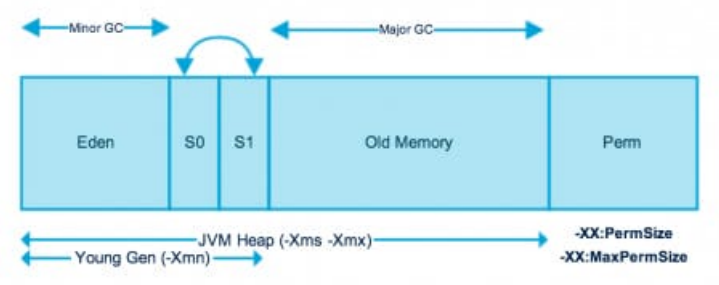
5.7 JVM 内存结构 本地方法栈

- 本地方法接口
  - 一个 Native Method 就是一个 Java 调用非 Java 代码的接口. A **native method** is a method that is implemented in a language other than Java. Natives methods are sometimes also referred to as **foreign methods**.
  - 为什么要使用 Native Method
    - 与 Java 环境外交互：有时 Java 应用需要与 Java 外面的环境交互，这就是本地方法存在的原因
    - 与操作系统交互：JVM 支持 Java 语言本身和运行时库，但是有时仍需要依赖一些底层系统的支持
    - 通过本地方法，我们可以实现用 Java 与实现了 jre 的底层系统交互， JVM 的一些部分就是 C 语言写的
- 本地方法栈（Native Method Stack）
  - JVM Stacks 用于管理 Java 方法的调用，而本地方法栈用于管理本地方法的调用
  - 本地方法栈也是线程私有的
  - 它的具体做法是 Native Method Stack 中登记 native 方法，在 Execution Engine 执行时加载本地方法库当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世界

5.8 JVM 内存结构 堆区

> **栈是运行时的单位，而堆是存储的单位**
>
> 栈解决程序的运行问题，即程序如何执行，或者说如何处理数据。堆解决的是数据存储的问题，即数据怎么放、放在哪

- Java 堆是 Java 虚拟机管理的内存中最大的一块，**被所有线程共享**, 也是垃圾收集器进行垃圾收集的最重要的内存区域
- 此内存区域的唯一目的就是**存放对象实例**，几乎所有的对象实例以及数据都在这里分配内存
- 因为 VM 采用分代收集算法，堆还可以细分为
  - 新生代(1/3 heap area)：Eden 区、From Survivor 区和 To Survivor 区
  - 老生代(2/3 heap area)：被长时间使用的对象，老生代的内存空间应该要比年轻代更大
- 分代的唯一理由就是优化 GC 性能
- 堆区大小通过参数 -Xms 和 -Xmx来设置，一般来说最小和最大设置为一样的，这样可以不用重新计算分区，可以节省一部分性能
- Java 虚拟机规范规定，Java 堆可以是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，像磁盘空间一样。实现时，既可以是固定大小，也可以是可扩展的，主流虚拟机都是可扩展的（通过 `-Xmx` 和 `-Xms` 控制），如果堆中没有完成实例分配，并且堆无法再扩展时，就会抛出 `OutOfMemoryError` 异常



- 年轻代 (Young Generation)
  - 年轻代是所有新对象创建的地方。当填充年轻代时，执行垃圾收集。这种垃圾收集称为 **Minor GC**
  - 年轻代分为伊甸园 (**Eden Memory**) 和两个幸存区 (**Survivor Memory**, 被称为from/to或s0/s1)，默认比例是`8:1:1`三个区
    - 大多数新创建的对象都位于 Eden 内存空间中
    - 当 Eden 空间被对象填充时，执行Minor GC，并将所有幸存者对象移动到一个幸存者空间中
    - Minor GC 检查幸存者对象，并将它们移动到另一个幸存者空间。所以每次，一个幸存者空间总是空的
    - 经过多次 GC 循环后存活下来的对象被移动到老生代, 默认情况下年龄到达 15 的对象会被移到老生代中
  - MinorGC 采用**复制算法** (复制->清空->互换)
    - eden、servicorFrom 复制到 ServicorTo，年龄+1
    - 清空 eden、servicorFrom
    - ServicorTo 和 ServicorFrom 互换
- 老生代(Old Generation)
  - 老生代主要存放应用程序中生命周期长的内存对象
  - 老生代gc称为Major GC，但对象比较稳定，所以 MajorGC 不会频繁执行。在进行 MajorGC 前一般都先进行了一次 MinorGC，使得有新生代的对象晋身入老生代，导致空间不够用时才触发
  - 无法找到足够大的连续空间分配给大对象会提前触发一次 MajorGC，直接进入老生代, 这样做的目的是避免在 Eden 区和两个Survivor 区之间发生大量的内存拷贝
  - MajorGC 采用**标记清除算法**
    - 首先扫描一次所有老生代，标记出存活的对象，然后回收没有标记的对象
    - MajorGC 的耗时比较长，因为要扫描再回收
    - MajorGC 会产生内存碎片，为了减少内存损耗，我们一般需要进行合并或者标记出来方便下次直接分配
    - 当老生代也满了装不下的时候，就会抛出 OOM（Out of Memory）异常
- 设置堆内存大小和 OOM
  - Java 堆用于存储 Java 对象实例，那么堆的大小在 JVM 启动的时候就确定了，我们可以通过 -Xmx 和 -Xms 来设定
  - -Xms 用来表示堆的起始内存，等价于 -XX:InitialHeapSize
    -Xmx 用来表示堆的最大内存，等价于 -XX:MaxHeapSize
  - 如果堆的内存大小超过 `-Xmx` 设定的最大内存， 就会抛出 `OutOfMemoryError` 异常

- 对象在堆中的生命周期
  - 在 JVM 内存模型的堆中，堆被划分为新生代和老生代
    - 新生代又被进一步划分为 Eden区 和 Survivor区，Survivor 区由 From Survivor 和 To Survivor 组成
  - 当创建一个对象时，对象会被优先分配到新生代的 Eden 区
    - 此时 JVM 会给对象定义一个**对象年轻计数器**（`-XX:MaxTenuringThreshold`）
  - 当 Eden 空间不足时，JVM 将执行新生代的垃圾回收（Minor GC）
    - JVM 会把存活的对象转移到 Survivor 中，并且对象年龄 +1
    - 对象在 Survivor 中同样也会经历 Minor GC，每经历一次 Minor GC，对象年龄都会+1
  - 如果分配的对象超过了-XX:PetenureSizeThreshold，对象会直接被分配到老生代
- 对象的分配过程
  - new 的对象先放在伊甸园区，此区有大小限制
  - 当伊甸园区的空间填满时，程序又需要创建对象，JVM 的垃圾回收器将对伊甸园区进行垃圾回收（Minor GC），将伊甸园区中的不再被其他对象所引用的对象进行销毁。再加载新的对象放到伊甸园区
  - 然后将伊甸园中的剩余对象移动到survivor0 区
  - 如果再次触发垃圾回收，gc伊甸园区以及survivor1区并放到survivor1区
  - 等到年龄到了一定程度（默认15次），可以通过-XX:MaxTenuringThreshold=设置，就放入老年区
  - 如果老年区内存不足, 执行Major GC
  - 如果之后还是放不下，就会产生 OOM 异常

5.9 逃逸分析

- 随着 JIT 编译期的发展和逃逸分析技术的逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化

  => **堆不是分配对象存储的唯一选择**
- **逃逸分析(Escape Analysis)** 是目前 Java 虚拟机中比较前沿的优化技术
  - 这是一种可以有效减少 Java 程序中同步负载和内存堆分配压力的跨函数全局数据流分析算法
- 通过逃逸分析，Java Hotspot 编译器能够分析出一个新的对象的引用的使用范围从而决定是否要将这个对象分配到堆上。逃逸分析的存在使得堆不一定是新对象的唯一去处，从而转向在栈上分配内存并创建对象，这样随着栈帧弹出对象也随即消失，**不需要GC。**逃逸分析就是来决定一个对象是否能分配在栈上的。如果在一个方法执行的过程中，有一个新建的对象没有被方法作用域之外的对象引用，我们就认为他不会逃逸
- 逃逸分析的基本行为就是分析对象动态作用域
  - 快速的判断是否发生了逃逸分析：**就看 new 的对象实体是否在方法外被调用**
  - 当一个对象在方法中被定义后，对象只在方法内部使用，则认为没有发生逃逸
  - 当一个对象在方法中被定义后，它被外部方法所引用，则认为发生逃逸。例如作为调用参数传递到其他地方中，称为方法逃逸

```
public static StringBuffer craeteStringBuffer(String s1, String s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb;
```

  - StringBuffer sb是一个方法内部变量，上述代码中直接将sb返回，这样这个 StringBuffer 有可能被其他方法所改变，这样它的作用域就不只是在方法内部，虽然它是一个局部变量，但是其逃逸到了方法外部。甚至还有可能被外部线程访问到，譬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

```
public static String createStringBuffer(String s1, String s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb.toString();
```

  - 不直接返回 StringBuffer，那么 StringBuffer 将不会逃逸出方法

- 参数设置
  - -XX: +DoEscapeAnalysis 开启逃逸分析
  - -XX: -DoEscapeAnalysis 关闭逃逸分析
  - -XX: +PrintEscapeAnalysis 查看逃逸分析结果
- 逃逸分析后如何优化
  - **栈上分配**：将堆分配转化为栈分配。如果一个对象在子程序中被分配，要使指向该对象的指针永远不会逃逸，对象可能是栈分配的候选，而不是堆分配
  - **同步省略**：如果一个对象被发现只能从一个线程被访问到，那么对于这个对象的操作可以不考虑同步

    线程同步的代价是相当高的，同步的后果是降低并发性和性能。在动态编译同步块的时候，`JIT` 编译器可以借助`逃逸分析`来判断同步块所使用的锁对象是否只能够被一个线程访问而没有被发布到其他线程，即该对象没有发生逃逸。如果没有逃逸，那么`JIT`编译器在编译这个同步块的时候就会取消对这部分代码的同步。这样就能大大提高并发性和性能。这个取消同步的过程就叫`同步省略`，也叫`锁消除`
  - **分离对象或标量替换**：有的对象可能不需要作为一个连续的内存结构存在也可以被访问到，那么对象的部分（或全部）可以不存储在内存，而存储在 CPU 寄存器

5.10 JVM 内存结构 方法区

- 方法区(Method Area) 也是元空间，是所有线程共享的内存区域
- 在 Java8 中永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代
  - Jdk7使用永久代，jdk8使用元空间
  - 永久代使用jvm的内存，而元空间不在虚拟机中，使用本地内存

- 运行时常量池(Runtime Constant Pool) 是方法区的一部分。Class 文件中除了有类的版本/字段/方法/接口等描述信息外，还有一项信息是常量池(Constant Pool Table)，用于存放编译期生成的各种字面量和符号引用，这部分内容将类在加载后进入方法区的运行时常量池中存放
- 永久代的内存回收的主要目标是针对常量池的回收和类型的卸载, 因此收益一般很小
- 方法区的大小和堆空间一样，可以选择固定大小也可选择可扩展，方法区的大小决定了系统可以放多少个类，如果系统类太多，导致方法区溢出，虚拟机同样会抛出内存溢出错误
- 方法区内部结构
  - 方法区用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等
- 运行时常量池
  - 常量池有三种： 运行时常量池，class常量池和字符串常量池
  - class 文件中的常量池包含
    - 字面量 -> String类型的字符串值或者定义为final类型的常量的值
    - 符号引用 -> 类或接口的全限定名， 变量或方法的名称， 变量或方法的描述信息，this
  - 在解释器解释执行每条JVM指令码的时候，根据这些指令码的`符号地址`去常量池中找到对应的描述
  - 运行时常量池就是将class文件中的常量池加载到内存当中，#x就会被转化为内存中的地址
  - 运行时常量池1.6放在方法区，也就是永久代。1.7放在永久代中，1.8改为放在元空间中（使用本地内存）

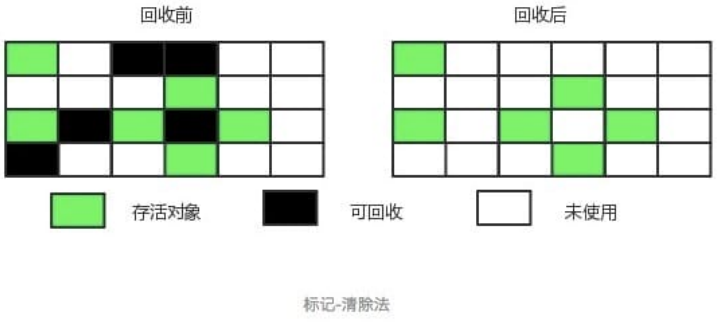
5.11 GC 判断对象可回收

- 引用计数算法 Reference Counting
  - 给对象添加一个引用计数器，当对象增加一个引用时计数器加 1，引用失效时计数器减 1。引用计数为 0 的对象可被回收
  - 两个对象出现**循环引用**的情况下，此时引用计数器永远不为 0，导致无法对它们进行回收
  - 正因为循环引用的存在，因此 Java 虚拟机不使用引用计数算法
- 可达性分析算法 GC Root Tracting
  - 为了解决引用计数法的循环引用问题，Java 使用了可达性分析的方法
  - 通过 GC Roots 作为起始点进行搜索，如果在“GC roots”和一个对象之间没有可达路径，则称该对象是不可达的
  - 要注意的是，不可达对象不等价于可回收对象，不可达对象变为可回收对象至少要经过两次标记过程
  - JVM 使用该算法来判断对象是否可被回收，在 Java 中 GC Roots 一般包含以下内容
    - 虚拟机栈中引用的对象
    - 本地方法栈中引用的对象
    - 方法区中类静态属性引用的对象
    - 方法区中的常量引用的对象
- 三色标记法
  - 通过三色标记可以理解为什么要STW
  - 白色：标识还未访问过

    - 黑色：标识已经被访问过，并且所有的引用已经扫描过且安全存活。如果有别的引用指向黑对象，则不需要继续扫描
    - 灰色：标识访问过，但是这个对象至少存在一个引用还没有被扫描过
- 方法区的回收
    - 方法区主要存放永久代对象，而永久代对象的回收率比新生代低很多，因此在方法区上进行回收性价比不高
    - 主要是对常量池的回收和对类的卸载
    - 常量的回收相对简单，没有引用就可以回收。但是类的卸载相对复杂，需要考虑三种情况：
        - 堆中不存在对象实例（包括子类对象）
        - 该类对应的 Class 对象没有在任何地方被引用，也就无法在任何地方通过反射访问该类方法
        - 类加载器已经卸载

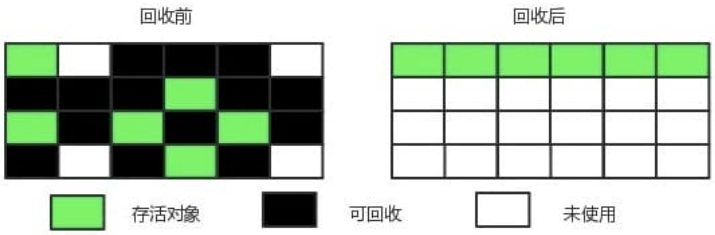
5.12 GC 垃圾回收算法

- **标记清除 Mark-Sweep**
    - 将存活的对象进行标记，然后清理掉未被标记的对象



标记-清除法

    - 同时需要两次遍历，第一次标记，第二次清除, 效率不够高
    - 会产生大量不连续的内存碎片，导致无法给大对象分配内存
    - **总结：效率低，内存碎片多**
- **复制 Copying**
    - 为了解决 Mark-Sweep 算法内存碎片化的缺陷而被提出的算法
    - 将内存划分为大小相等的两块，每次只使用其中一块，当这一块内存用完了就将还存活的对象复制到另一块上面，然后再把使用过的内存空间进行一次清理
    - 回收新生代就是这种收集算法
    - **总结：优点不会出现大量内存碎片，缺点是空间没有合理利用**
- **标记整理 Mark-Compack**
    - 标记阶段和 Mark-Sweep 算法相同，标记后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



- 分代收集 Generational Collection
  - 分代收集法是目前大部分 JVM 所采用的方法，其核心思想是根据对象存活的不同生命周期将内存划分为不同的域，一般情况下将 GC 堆划分为新生代(Young Generation)和老生代(Tenured/Old Generation)，因此可以根据不同区域选择不同的算法
  - 新生代
    - 新生代的特点是每次垃圾回收时都有大量垃圾需要被回收，即要复制的操作比较少
    - 通常并不是按照 1：1 来划分新生代。一般将新生代划分为一块较大的 Eden 空间和两个较小的 Survivor 空间(From Space, To Space)，每次使用Eden 空间和其中的一块 Survivor 空间，当进行回收时，将该两块空间中还存活的对象复制到另一块 Survivor 空间中
    - **新生代使用复制算法**
  - 老生代
    - 老生代的特点是每次垃圾回收时只有少量对象需要被回收
    - **老生代使用标记整理算法**
- 回收策略 Minor GC、Major GC、Full GC
  - JVM 在进行 GC 时，并非每次都对堆内存（新生代、老生代；方法区）区域一起回收的，大部分时候回收的都是指新生代。
  - 针对 HotSpot VM 的实现，它里面的 GC 按照回收区域又分为两大类：部分收集（Partial GC），整堆收集（Full GC）
  - 部分收集：不是完整收集整个 Java 堆的垃圾收集。其中又分为：
    - 新生代收集（Minor GC/Young GC）：只是新生代的垃圾收集
    - 老生代收集（Major GC/Old GC）：只是老生代的垃圾收集
      - 目前，只有 CMS GC 会有单独收集老生代的行为
      - 很多时候 Major GC 会和 Full GC 混合使用，需要具体分辨是老生代回收还是整堆回收
    - 混合收集（Mixed GC）：收集整个新生代以及部分老生代的垃圾收集
      - 目前只有 G1 GC 会有这种行为
  - 整堆收集（Full GC）：收集整个 Java 堆和方法区的垃圾

5.13 GC 引用类型

- 无论是通过引用计算算法判断对象的引用数量，还是通过可达性分析算法判断对象是否可达，判定对象是否可被回收都与引用有关
- Java 具有四种强度不同的引用类型

- 强引用
  - 使用 new 一个新对象的方式或者把一个对象赋给一个引用变量，这个引用变量就是一个强引用

    ```
    Object object = new Object();
    ```

  - 被强引用关联的对象处于可达状态，不会被垃圾回收机制回收，即使该对象以后不会被用到 JVM 也不会回收，哪怕是OOM
  - 因此强引用是造成 Java 内存泄漏的主要原因之一
- 软引用
  - 使用 SoftReference 类来创建软引用

    ```
    SoftReference<String> stringSoftReference=new SoftReference<>("sss");
    ```

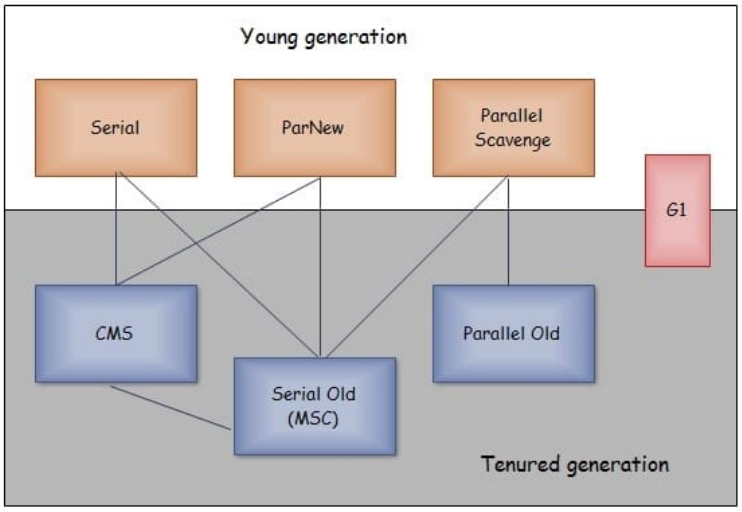
  - 被软引用关联的对象当系统内存足够时不会被回收，只有在内存不够的情况下才会被回收
  - 软引用通常用在对内存敏感的程序中, 使用软引用能防止内存泄露，增强程序的健壮性
- 弱引用
  - 使用 WeakReference 类来实现弱引用

    ```
    WeakReference<String> weakReference=new WeakReference<>("aaae");
    ```

  - 被弱引用关联的对象一定会被回收，也就是说它只能存活到下一次垃圾回收发生之前
  - 弱引用比软引用的生存期更短，对于只有弱引用的对象来说，只要垃圾回收机制一运行，不管 JVM 的内存空间是否足够，总会回收该对象占用的内存
- 虚引用
  - 使用 PhantomReference 来实现虚引用
  - 不能单独使用，必须和引用队列联合使用
  - 一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用取得一个对象
  - 虚引用的主要作用是跟踪对象被垃圾回收的状态

5.14 GC 垃圾收集器

- Java 堆内存被划分为新生代和年老代两部分，新生代主要使用复制和标记-清除垃圾回收算法；老生代主要使用标记-整理垃圾回收算法，因此 java 虚拟中针对新生代和老生代分别提供了多种不同的垃圾收集器
- JDK中 Sun HotSpot 虚拟机中有7个垃圾收集器，连线表示垃圾收集器可以配合使用



- 单线程与多线程:
  - 单线程指的是垃圾收集器只使用一个线程进行收集
  - 多线程使用多个线程进行收集
- 串行与并行:
  - 串行指的是垃圾收集器与用户程序交替执行，这意味着在执行垃圾收集的时候需要停顿用户程序;
  - 并行指的是垃圾收集器和用户程序同时执行
  - 除了 CMS 和 G1 之外，其它垃圾收集器都是以串行的方式执行
- **Serial 收集器 (单线程、串行、复制算法)**
  - 只会使用一个 CPU 或一条线程去完成gc，并且在进行垃圾收集的同时，必须暂停其他所有的工作线程，直到垃圾收集结束
  - 优点是简单高效，对于单个 CPU 环境来说，由于没有线程交互的开销，因此拥有最高的单线程收集效率
  - Serial垃圾收集器是 java 虚拟机运行在 Client 模式下新生代的默认垃圾收集器
- **ParNew 收集器 (多线程、串行、复制算法)**
  - 它是 Serial 收集器的多线程版本，垃圾收集过程中同样也要暂停所有其他的工作线程
  - ParNew 收集器默认开启和 CPU 数目相同的线程数，可以通过-XX:ParallelGCThreads 参数来限制垃圾收集器的线程数
  - ParNew 垃圾收集器是java虚拟机运行在 Server 模式下新生代的默认垃圾收集器
- **Parallel Scavenge 收集器 (多线程、串行、复制算法)**
  - 其它收集器关注点是尽可能缩短垃圾收集时用户线程的停顿时间，而它的目标是达到一个可控制的吞吐量Thoughput，它被称为"吞吐量优先"收集器, 这里的吞吐量指 CPU 用于运行用户代码的时间/CPU 总消耗时间
    - 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)
  - 停顿时间之于吞吐量
    - 停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验

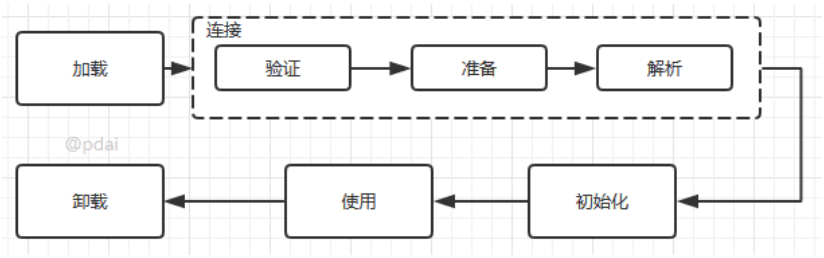
        - 吞吐量越高就越可以高效率地利用 CPU 时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务
        - 缩短停顿时间是以牺牲吞吐量和新生代空间来换取的: 新生代空间变小，垃圾回收变得频繁，导致吞吐量下降
    - 自适应调节策略也是 ParallelScavenge 收集器与 ParNew 收集器的一个重要区别
- **Serial Old 收集器 (单线程、串行、标记整理算法)**
    - Serial Old 是 Serial 垃圾收集器的老生代版本，新生代采取标记复制，老生代采取标记整理算法
    - Serial old在Server模式下主要有两个用途：
        - 在 JDK1.5 之前版本中与Parallel Scavenger进行配合回收
        - 作为CMS 收集器的后备预案
    - Serial Old 垃圾收集器运行在 Client 默认的 java 虚拟机的默认老生代垃圾收集器
- **Parallel Old 收集器 (多线程、串行、复制算法)**
    - Parallel Old 是 Parallel Scavenge 收集器的老生代版本，老生代使用标记整理算法
    - 如果系统对吞吐量要求比较高，可以优先考虑新生代 Parallel Scavenge 加老生代 Parallel Old 收集器搭配策略
- **CMS 收集器 (多线程、并行、标记清除算法)**
    - CMS(Concurrent Mark Sweep) 是老生代垃圾收集器，其最主要目标是获取最短垃圾回收停顿时间，和其他老生代使用标记整理算法不同，它使用多线程的标记清除算法。最短的垃圾收集停顿时间可以为交互比较高的程序提高用户体验
    - CMS 工作机制相比其他的垃圾收集器来说更复杂，分为以下四个流程：
        - 初始标记: 仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要停顿STW
        - 并发标记: 进行 GC Roots Tracing 的过程，它在整个回收过程中耗时最长，和用户线程一起工作，不需要停顿
            - 产生两种问题：错标 - 通过重新标记解决 、漏标 - 产生浮动垃圾
        - 重新标记: 为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，需要停顿STW
        - 并发清除: 清除 GC Roots 不可达对象，和用户线程一起工作，不需要停顿暂停所有工作线程
    - 整个过程中耗时最长的**并发标记**和**并发清除**过程中，收集器线程都可以与用户线程一起工作，不需要进行停顿，所以总体上来看CMS 收集器的内存回收和用户线程是一起并发地执行
    - 缺点:
        - 吞吐量低: 低停顿时间是以牺牲吞吐量为代价的，导致 CPU 利用率不够高
        - 无法处理浮动垃圾，可能出现 Concurrent Mode Failure. 浮动垃圾是指并发清除阶段由于用户线程继续运行而产生的垃圾，这部分垃圾只能到下一次 GC 时才能进行回收
        - 标记清除算法导致的空间碎片，往往出现老生代空间剩余，但无法找到足够大连续空间来分配当前对象，不得不提前触发一次 Full GC
- **G1 收集器 (标记整理算法)**
    - G1(Garbage-First)，它是一款面向服务端应用的垃圾收集器，在多 CPU 和大内存的场景下有很好的性能

- 相比与 CMS 收集器，G1 收集器两个最突出的改进是：
  - 基于标记整理算法，不产生内存碎片
  - 可以非常精确控制停顿时间，在不牺牲吞吐量前提下，实现低停顿垃圾回收
- 堆被分为新生代和老生代，其它收集器进行收集的范围都是整个新生代或者老生代，而 G1 可以直接对新生代和老生代一起回收，同时G1 收集器避免全区域垃圾收集，它把堆内存划分为大小固定的几个独立区域，并且跟踪这些区域的垃圾收集进度，同时在后台维护一个优先级列表，每次根据所允许的收集时间，优先回收垃圾最多的区域。区域划分和优先级区域回收机制，确保 G1 收集器可以在有限时间获得最高的垃圾收集效率

- JDK8 默认垃圾收集器是Parallel Scavenge+Parallel Old

5.15 JVM 类的生命周期

- JVM类的加载的过程分为五个部分：`加载`、`验证`、`准备`、`解析`、`初始化`，其中`连接`有三个步骤`验证`、`准备`、`解析`



- **类的加载: 查找并加载类的二进制数据**
  - 这个阶段会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入口
  - 完成以下三件事情
    - 通过一个类的全限定名来获取其定义的二进制字节流
    - 将字节流代表的静态储存结构转化为方法区中的运行时数据结构
    - 在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口
- **类的验证: 确保被加载的类的正确性**
  - 这个阶段目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全
  - 验证阶段大致会完成4个阶段
    - `文件格式验证`: 验证字节流是否符合Class文件格式的规范
    - `元数据验证`: 对字节码描述的信息进行语义分析, 保证其描述的信息符合Java语言规范的要求
    - `字节码验证`: 通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的
    - `符号引用验证`: 确保解析动作能正确执行
- **类的准备: 为类的静态变量分配内存，并将其初始化为默认值**
  - 这个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，即在方法区中分配这些变量所使用的内存空间

    - 进行内存分配的**仅包括**类变量(`static`)，而**不包括**实例变量，实例变量会在对象实例化时随着对象一块分配在Java堆中
    - 这里所设置的初始值通常情况下是数据类型默认的零值(如`0`、`0L`、`null`、`false`等)
        - 假设类变量的定义为: `int value = 3`；那么变量value在准备阶段过后的初始值为`0`，而不是`3`
        - 假设类变量的定义为: `final int value = 3`；准备阶段虚拟机就会根据ConstantValue属性设置将value赋值为3
    - 总结：分配内存，还没有赋值，都是默认值（final除外）
- **类的解析: 把类中的符号引用转换为直接引用**
    - 这个阶段是虚拟机将常量池内的符号引用替换为直接引用，之前使用符号引用是因为还没有分配内存，不知道地址在哪里
    - 解析动作主要针对`类`或`接口`、`字段`、`类方法`、`接口方法`、`方法类型`、`方法句柄`和`调用点`
    - 符号引用
        - 符号引用与虚拟机实现的布局无关，引用的目标并不一定要已经加载到内存中
        - 各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在 Java 虚拟机规范的 Class 文件格式中
    - 直接引用
        - 直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄
        - 如果有了直接引用，那引用的目标必定已经在内存中存在
- **类的初始化**
    - 这个阶段为类的静态变量赋予正确的初始值，JVM负责对类进行初始化，主要对类变量进行初始化
    - 在Java中对类变量进行初始值设定有两种方式
        - 声明类变量是指定初始值
        - 使用静态代码块为类变量指定初始值
    - 类初始化时机
        - 创建类的实例，也就是new的方式，如果类没有进行过初始化
        - 访问某个类或接口的静态变量，或者对该静态变量赋值
        - 调用类的静态方法
        - 使用 java.lang.reflect包的方法对类进行反射调用，如果类没有进行过初始化
        - 初始化某个类的子类，则其父类也会被初始化
- 使用
    - 类访问方法区内的数据结构的接口， 对象是Heap区的数据
- 类的卸载
    - 类的所有实例被回收，也就是java堆中不存在该类的任何实例
    - 加载该类的ClassLoader已经被回收
    - 该类对应的java.lang.Class对象没有被引用，无法在任何地方通过反射访问该类的方法
- Java虚拟机将结束生命周期的几种情况

- 执行了System.exit()方法
- 程序正常执行结束
- 程序在执行过程中遇到了异常或错误而异常终止
- 由于操作系统出现错误而导致Java虚拟机进程终止

5.16 类加载器

- 对于任意一个类，都必须有加载他的类加载器和这个类本身一起确立在JVM中的唯一性
  - 换句话说，比较两个类相等的时候，必须是由同一个类加载器加载为前提。否则就是不相等
- 类加载器可以大致划分为以下三类
  - 启动类加载器 (Bootstrap ClassLoader): 负责加载存放在 `JAVA_HOME\lib` 下，或被 `-Xbootclasspath` 参数指定的路径中的，并且能被虚拟机识别的类库(如rt.jar，所有的java.*开头的类均被Bootstrap ClassLoader加载)
  - 扩展类加载器 (Extension ClassLoader): 它负责加载 `JAVA_HOME\lib\ext` 目录中，或者由 `java.ext.dirs` 系统变量指定的路径中的所有类库(如javax.*开头的类)，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): 它负责加载用户类路径(ClassPath)所指定的类，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
- 类加载有三种方式
  - 命令行启动应用时候由JVM初始化加载
  - 通过Class.forName()方法动态加载:
    - 将类的.class文件加载到jvm中之外，还会对类进行解释，执行类中的static块
  - 通过ClassLoader.loadClass()方法动态加载:
    - 只将.class文件加载到jvm中, 不会执行static中的内容,只有在newInstance才会去执行static块

5.17 JVM 类加载机制

- `全盘负责`，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入
- `父类委托`，先让父类加载器试图加载该类，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类
- `缓存机制`，缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效
- `双亲委派机制`, 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，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类

5.18 JVM 双亲委派

- 当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父类去完成，每一个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载其中，只有当父类加载器反馈自己无法完成这个请求的时候（在它的加载路径下没有找到所需加载的Class），子类加载器才会尝试自己去加载
- 采用双亲委派的一个好处是比如加载位于 rt.jar 包中的类 java.lang.Object，不管是哪个加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载
    - 保证了使用不同的类加载器最终得到的都是同样一个 Object 对象

缓存池 (buffer pool)

- 应用系统分层架构，为了加速数据访问，会把最常访问的数据，放在**缓存**(cache)里，避免每次都去访问数据库
- 操作系统，会有**缓冲池**(buffer pool)机制，避免每次访问磁盘，以加速数据的访问
- new Integer(123) 与 Integer.valueOf(123) 的区别在于:
    - new Integer(123) 每次都会新建一个对象
    - Integer.valueOf(123) 会使用缓存池中的对象，多次调用会取得同一个对象的引用。

```
Integer x = new Integer(123);
Integer y = new Integer(123);
System.out.println(x == y);    // false
Integer z = Integer.valueOf(123);
Integer k = Integer.valueOf(123);
System.out.println(z == k);   // true
```

- valueOf() 方法的实现比较简单，就是先判断值是否在缓存池中，如果在的话就直接返回缓存池的内容。
- Integer 缓存池的大小默认为 -128~127
- 基本类型对应的缓冲池
    - boolean values true and false
    - all byte values
    - short values between -128 and 127
    - int values between -128 and 127
    - char in the range \u0000 to \u007F
    - **最大值 127 可以通过 JVM 的启动参数 -XX:AutoBoxCacheMax=size 修改**

https://blog.csdn.net/w20001118/article/details/125724647

6. ♥ Java Design Patterns ♥

5.1 Design Patterns Concept

- "设计模式"最初并不是出现在软件设计中，而是被用于建筑领域的设计中
- 软件设计模式（Software Design Pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结
- 它描述了在软件设计过程中的一些不断重复发生的问题，以及该问题的解决方案。也就是说，它是解决特定问题的一系列套路，是前辈们的代码设计经验的总结，具有一定的普遍性，可以反复使用

5.2 设计模式分类

- **创建型模式 Creational patterns**
  - 用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”
  - 单例、原型、工厂方法、抽象工厂、建造者等 5 种创建型模式。
- **结构型模式 Structural Patterns**
  - 用于描述如何将类或对象按某种布局组成更大的结构
  - 代理、适配器、桥接、装饰、外观、享元、组合等 7 种结构型模式。
- **行为型模式 Behavioral Patterns**
  - 用于描述类或对象之间怎样相互协作共同完成单个对象无法单独完成的任务，以及怎样分配职责
  - 模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种行为型模式

5.3 UML图

- 统一建模语言（Unified Modeling Language，UML）是用来设计软件的可视化建模语言
- 从目标系统的不同角度出发，定义了用例图、类图、对象图、状态图、活动图、时序图、协作图、构件图、部署图等 9 种图
- 类的表示方式
  - 类使用包含类名、属性(field) 和方法(method) 且带有分割线的矩形来表示
  - 加号和减号表示了这个属性/方法的可见性: +表示public : 表示private #表示protected
  - 属性的完整表示方式是： **可见性 名称： 类型 [= 缺省值]**
  - 方法的完整表示方式是： **可见性 名称(参数列表) [： 返回类型]**
- 类与类之间关系表示
  - 关联关系: 关联又可以分为单向关联，双向关联，自关联
  - 聚合关系: 关联关系的一种，是强关联关系，是**整体和部分**之间的关系
  - 组合关系: 表示类之间的整体与部分的关系，但它是一种更强烈的聚合关系
  - 依赖关系: 表示一种使用关系，它是对象之间耦合度最弱的一种关联方式，是临时性的关联
  - 继承关系: 对象之间耦合度最大的一种关系，表示一般与特殊的关系，是父类与子类之间的关系，是一种继承关系
  - 实现关系: 是接口与实现类之间的关系。在这种关系中，类实现了接口，类中的操作实现了接口中所声明的所有的抽象操作

5.4 软件设计原则

- **Single Responsibility**
  - 一个接口只负责一件事情，只能有一个原因导致类变化
- **Open for Extension, Closed for Modification**
  - 在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果，是为了使程序的扩展性好，易于维护和升级
  - 想要达到这样的效果，我们需要使用接口和抽象类
- **Liskov Substitution**
  - 任何基类可以出现的地方，子类一定可以出现。子类可以扩展父类的功能，但不能改变父类原有的功能
  - 如果通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的概率会非常大
- **Interface Segregation**
  - 客户端不应该被迫依赖于它不使用的方法；一个类对另一个类的依赖应该建立在最小的接口上
- **Dependency Inversion**
  - 高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象
  - 简单说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合
- **迪米特法则**
  - 只和朋友交流（成员变量、方法输入输出参数），不和陌生人说话，控制好访问修饰符
  - 如果两个对象无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用
  - 其目的是降低类之间的耦合度，提高模块的相对独立性
- **Composite/Aggregate Reuse Principle**
  - 合成复用原则是指尽量使用对象组合(has-a)/聚合(contanis-a)达到软件复用的目的，其次才使用继承关系
  - 通常类的复用分为继承复用和合成复用两种
  - 继承复用虽然有简单和易实现的优点，但它也存在以下缺点
    - 继承复用破坏了类的封装性。因为继承会将父类的实现细节暴露给子类，父类对子类是透明的 (白箱"复用)
    - 子类与父类的耦合度高。父类的实现的任何改变都会导致子类的实现发生变化，这不利于类的扩展与维护
    - 限制了复用的灵活性。从父类继承而来的实现是静态的，在编译时已经定义，所以在运行时不可能发生变化
  - 组合或聚合复用时，可以吸纳新对象, 可以调用已有对象的功能，它有以下优点
    - 维持了类的封装性。因为成分对象的内部细节是新对象看不见的 ("黑箱"复用)
    - 对象间的耦合度低。可以在类的成员位置声明抽象

        - 复用的灵活性高。这种复用可以在运行时动态进行，新对象可以动态地引用与成分对象类型相同的对象

5.5 创建者模式

- **创建者模式 (Creational patterns)** 主要关注“怎样创建对象”，特点是“将对象的创建与使用分离”，这样可以降低系统的耦合度，使用者不需要关注对象的创建细节
- **单例模式 Singleton Pattern**
    - 涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建
    - 这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象
    - 保证一个类仅有一个static实例，并提供一个访问它的全局访问点
    - 破坏单例模式
        - 使用序列化和反射可以创建多个对象
    - 解决方法
        - 序列化破坏单例模式解决方法: Singleton类中添加 `readResolve()` 方法
        - 反序列方式破坏单例模式解决方法: 再私有构造方法中添加 `if(instance != null) throw new RuntimeException();`
- **工厂方法模式 Factory Method Pattern**
    - 在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象
    - 定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行
    - 在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，满足开闭原则
    - 每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这**增加了系统的复杂度**
- **抽象工厂模式 Abstract Factory Pattern**
    - 抽象工厂模式是围绕一个超级工厂创建其他工厂, 该超级工厂又称为其他工厂的工厂
    - 接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象
    - 提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类
    - 抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品
    - 使用场景
        - 当需要创建的对象是一系列相互关联或相互依赖的产品族时
        - 系统中有多个产品族，但每次只使用其中的某一族产品
        - 系统中提供了产品的类库，且所有产品的接口相同，客户端不依赖产品实例的创建细节和内部结构
    - Collection获取iterator就是工厂方法模式
        - Collection接口是抽象工厂，ArrayList是具体工厂，Iterator接口是抽象商品，ArrayListIter是具体产品

- **原型模式 Prototype Pattern**
  - 原型模式是用于创建当前对象的克隆，同时又能保证性能，当直接创建对象的代价比较大时，则采用这种模式
  - 用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型对象相同的新对象
  - 原型模式包含如下角色：
    - 抽象原型类：规定了具体原型对象必须实现的的 clone() 方法
    - 具体原型类：实现抽象原型类的 clone() 方法，它是可被复制的对象
    - 访问类：使用具体原型类中的 clone() 方法来复制新的对象
  - 原型模式的克隆分为浅克隆和深克隆
    - 浅克隆：创建一个新对象，新对象的属性和原来完全相同，对于非基本类型属性，仍指向原有属性所指向的对象的内存地址
    - 深克隆：创建一个新对象，属性中引用的其他对象也会被克隆，不再指向原有对象地址
  - Java中的Object类中提供了 `clone()` 方法来实现浅克隆，Cloneable 接口是抽象原型类，而实现了Cloneable接口的子实现类就是具体的原型类
  - 序列化+反序列化 实现深拷贝

    必须实现Serializable接口，否则会抛NotSerializableException异常

    ```
    //创建对象输出流对象
    ObjectOutputStream oos = new ObjectOutputStream(new
    FileOutputStream("C:\\Users\\Think\\Desktop\\b.txt"));
    //将c1对象写出到文件中
    oos.writeObject(c1);
    oos.close();

    //创建对象出入流对象
    ObjectInputStream ois = new ObjectInputStream(new
    FileInputStream("C:\\Users\\Think\\Desktop\\b.txt"));
    //读取对象
    Citation c2 = (Citation) ois.readObject();
    ```

- **建造者模式 Builder Pattern**
  - 建造者模式使用多个简单的对象一步一步构建成一个复杂的对象
  - 将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示
  - 分离了部件的构造(由Builder来负责)和装配(由Director负责)，从而可以构造出复杂的对象，这个模式适用于某个对象的构建过程复杂的情况
  - 由于实现了构建和装配的解耦，不同的构建器，相同的装配，也可以做出不同的对象；相同的构建器，不同的装配顺序也可以做出不同的对象。也就是实现了构建算法、装配算法的解耦，实现了更好的复用
  - 建造者模式可以将部件和其组装过程分开，一步一步创建一个复杂的对象。用户只需要指定复杂对象的类型就可以得到该对象，而无须知道其内部的具体构造细节
  - 建造者（Builder）模式包含如下角色：

        - 抽象建造者类(Builder): 这个接口规定要实现复杂对象的那些部分的创建，并不涉及具体的部件对象的创建
        - 具体建造者类(ConcreteBuilder): 实现 Builder 接口，完成复杂产品的各个部件的具体创建方法。在构造过程完成后，提供产品的实例
        - 产品类(Product): 要创建的复杂对象
        - 指挥者类(Director): 调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建
    - 优缺点
        - 建造者模式的封装性很好。使用建造者模式可以有效的封装变化，在使用建造者模式的场景中，一般产品类和建造者类是比较稳定的，因此，将主要的业务逻辑封装在指挥者类中对整体而言可以取得比较好的稳定性
        - 在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象
        - 可以更加精细地控制产品的创建过程 。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程
        - 建造者模式很容易进行扩展。如果有新的需求，通过实现一个新的建造者类就可以完成，基本上不用修改之前已经测试通过的代码，因此也就不会对原有功能引入风险。符合开闭原则
        - 造者模式所创建的产品一般具有较多的共同点，其组成部分相似，**如果产品之间的差异性很大，则不适合使用建造者模式**，因此其使用范围受到一定的限制
- 创建者模式对比
    - 工厂方法模式VS建造者模式
        - 工厂方法模式注重的是整体对象的创建方式；而建造者模式注重的是部件构建的过程，意在通过一步一步地精确构造创建出一个复杂的对象
    - 抽象工厂模式VS建造者模式
        - 抽象工厂模式实现对产品家族的创建，一个产品家族是这样的一系列产品：具有不同分类维度的产品组合，采用抽象工厂模式则是不需要关心构建过程，只关心什么产品由什么工厂生产即可; 建造者模式则是要求按照指定的蓝图建造产品，它的主要目的是通过组装零配件而产生一个新产品

5.6 结构型模式

- **结构型模式 (Structural Patterns)** 描述如何将类或对象按某种布局组成更大的**结构**
    - 它分为类结构型模式和对象结构型模式
    - 前者采用继承机制来组织接口和类，后者采用组合或聚合来组合对象
- **代理模式 Proxy Pattern**
    - 由于某些原因需要给某对象提供一个代理, 以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介
    - Java中的代理按照代理类生成时机不同又分为静态代理和动态代理。静态代理代理类在编译期就生成，而动态代理代理类则是在Java运行时动态生成。动态代理又有JDK代理和CGLib代理两种
    - 代理模式分为三种角色:

        - 抽象主题(Subject) 类： 通过接口或抽象类声明真实主题和代理对象实现的业务方法
        - 真实主题 (Real Subject) 类： 实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象
        - 代理 (Proxy) 类 ： 提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能
    - 三种代理的对比
        - 动态代理和静态代理:

          动态代理与静态代理相比较，最大的好处是接口中声明的所有方法都被转移到调用处理器一个集中的方法中处理（InvocationHandler.invoke）。这样在接口方法数量比较多的时候，我们可以进行灵活处理，而不需要像静态代理那样每一个方法进行中转

          如果接口增加一个方法，静态代理模式除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。而动态代理不会出现该问题
        - jdk代理和CGLIB代理:

          使用CGLib实现动态代理，CGLib底层采用ASM字节码生成框架，使用字节码技术生成代理类，在JDK1.6之前比使用Java反射效率要高。唯一需要注意的是，CGLib不能对声明为final的类或者方法进行代理，因为CGLib原理是动态生成被代理类的子类。

          在JDK1.6、JDK1.7、JDK1.8逐步对JDK动态代理优化之后，在调用次数较少的情况下，JDK代理效率高于CGLib代理效率，只有当进行大量调用的时候，JDK1.6和JDK1.7比CGLib代理效率低一点，但是到JDK1.8的时候，JDK代理效率高于CGLib代理。所以如果有接口使用JDK动态代理，如果没有接口使用CGLIB代理
    - 优点:
        - 代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用
        - 代理对象可以扩展目标对象的功能
        - 代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度
    - 缺点:
        - 由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢
        - 实现代理模式需要额外的工作，有些代理模式的实现非常复杂
- **适配器模式 Adapter Pattern**
    - 适配器模式是作为两个不兼容的接口之间的桥梁，它结合了两个独立接口的功能
    - 将一个类的接口转换成客户希望的另外一个接口。适配器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
    - 适配器模式分为类适配器模式和对象适配器模式
        - 前者类之间的耦合度比后者高，且要求程序员了解现有组件库中的相关组件的内部结构，所以应用相对较少些
    - 适配器模式（Adapter）包含以下主要角色
        - 目标 (Target) 接口：当前系统业务所期待的接口，它可以是抽象类或接口
        - 适配者 (Adaptee) 类：它是被访问和适配的现存组件库中的组件接口

        - 适配器 (Adapter) 类：它是一个转换器，通过继承或引用适配者的对象，把适配者接口转换成目标接口，让客户按目标接口的格式访问适配者
        - Adaptee Class <- Adapter class <- Target Interface
    - 类适配器模式
        - 定义一个适配器类来实现当前系统的业务接口，同时又继承现有组件库中已经存在的组件
        - 类适配器模式违背了合成复用原则。类适配器是客户类有一个接口规范的情况下可用，反之不可用
    - 对象适配器模式
        - 对象适配器模式可采用将现有组件库中已经实现的组件引入适配器类中，该类同时实现当前系统的业务接口
        - 传入适配者，并在适配器里面直接叫适配者的方法
- **装饰者模式 Decorator Pattern**
    - 装饰器模式允许向一个现有的对象添加新的功能，同时又不改变其结构
    - 这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能
    - 装饰 (Decorator) 模式中的角色：
        - 抽象构件(Component) 角色 ： 定义一个抽象接口以规范准备接收附加责任的对象
        - 具体构件 (Concrete Component) Concrete Component角色 ： 实现抽象构件，通过装饰角色为其添加一些职责
        - 抽象装饰 (Decorator) 角色 ： 继承或实现抽象构件，并包含具体构件的实例，可以通过其子类扩展具体构件的功能
        - 具体装饰 (ConcreteDecorator) 角色 ： 实现抽象装饰的相关方法，并给具体构件对象添加附加的责任
    - 好处
        - 饰者模式可以带来比继承更加灵活性的扩展功能，使用更加方便，可以通过组合不同的装饰者对象来获取具有不同行为状态的多样化的结果。装饰者模式比继承更具良好的扩展性，完美的遵循开闭原则，继承是静态的附加责任，装饰者则是动态的附加责任
        - 装饰类和被装饰类可以独立发展不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能
    - IO流中的包装类使用到了装饰者模式: BufferedInputStream，BufferedOutputStream，BufferedReader，BufferedWrite

      ```
      FileWriter fw = new FileWriter("a.txt");
      BufferedWriter bw = new BufferedWriter(fw);
      ```

        - BufferedWriter使用装饰者模式对Writer子实现类进行了增强，添加了缓冲区，提高了写数据的效率
    - 静态代理和装饰者模式的区别
        - 相同点
            - 都要实现与目标类相同的业务接口

        - 在两个类中都要声明目标对象
        - 都可以在不修改目标类的前提下增强目标方法
    - 不同点
        - **目的不同**: 装饰者是为了增强目标对象; 静态代理是为了保护和隐藏目标对象
        - 获取目标对象构建来源不同: 装饰者是由外界传递进来，可以通过构造方法传递; 静态代理是在代理类内部创建，以此来隐藏目标对象

- **桥接模式 Bridge Pattern**
    - 桥接模式是用于把抽象化与实现化解耦，使得二者可以独立变化
    - 它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度
    - 桥接模式包含以下主要角色：
        - 抽象化 (Abstraction) 角色 ： 定义抽象类，并包含一个对实现化对象的引用
        - 扩展抽象化 (Refined Abstraction) 角色：是抽象化角色的子类，实现父类中的业务方法，并通过组合关系调用实现化角色中的业务方法
        - 实现化 (Implementer) 角色: 定义实现化角色的接口，供扩展抽象化角色调用
        - 具体实现化 (Concrete Implementer) 角色: 给出实现化角色接口的具体实现
    - 优点
        - 桥接模式提高了系统的可扩充性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统
        - 实现细节对客户透明
    - 使用场景
        - 当一个类存在两个独立变化的维度，且这两个维度都需要进行扩展时
        - 当一个系统不希望使用继承或因为多层次继承导致系统类的个数急剧增加时当一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性时, 避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系

- **外观模式 Facade Pattern**
    - 外观模式隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口
    - 为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得子系统更加容易使用
    - 外观 (Facade) 模式包含以下主要角色:
        - 外观 (Facade) 角色：为多个子系统对外提供一个共同的接口
        - 子系统 (Sub System) 角色：实现系统的部分功能，客户可以通过外观角色访问它
    - 优点
        - 降低了子系统与客户端之间的耦合度，使得子系统的变化不会影响调用它的客户类
        - 对客户屏蔽了子系统组件，减少了客户处理的对象数目，并使得子系统使用起来更加容易, 提高灵活性, 提高了安全性
    - 缺点
        - 不符合开闭原则，修改很麻烦，继承重写都不合适

- **组合模式 Composite Pattern**
  - 组合模式，又叫部分整体模式，是用于把一组相似的对象当作一个单一的对象
  - 将对象组合成树形结构以表示"部分-整体"的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性
  - 组合模式主要包含三种角色:
    - 抽象根节点 (Component): 定义系统各层次对象的共有方法和属性，可以预先定义一些默认行为和属性
    - 树枝节点 (Composite): 定义树枝节点的行为，存储子节点，组合树枝节点和叶子节点形成一个树形结构
    - 叶子节点 (Leaf): 叶子节点对象，其下再无分支，是系统层次遍历的最小单位
  - 组合模式分为透明组合模式和安全组合模式两种形式
    - 透明组合模式: 透明组合模式中，抽象根节点角色中声明了所有用于管理成员对象的方法，透明组合模式也是组合模式的标准形式。透明组合模式的缺点是不够安全，因为叶子对象和容器对象在本质上是有区别的，叶子对象不可能有下一个层次的对象，即不可能包含成员对象
    - 安全组合模式: 在安全组合模式中，在抽象构件角色中没有声明任何用于管理成员对象的方法，而是在树枝节点中声明并实现这些方法。安全组合模式的缺点是不够透明，因为叶子构件和容器构件具有不同的方法，且容器构件中那些用于管理成员对象的方法没有在抽象构件类中定义，因此客户端不能完全针对抽象编程，必须有区别地对待叶子构件和容器构件
  - 优点
    - 组合模式可以清楚地定义分层次的复杂对象，表示对象的全部或部分层次，它让客户端忽略了层次的差异，方便对整个层次结构进行控制
    - 客户端可以一致地使用一个组合结构或其中单个对象，不必关心处理的是单个对象还是整个组合结构，简化了客户端代码
    - 在组合模式中增加新的树枝节点和叶子节点都很方便，无须对现有类库进行任何修改，符合“开闭原则”
    - 组合模式为树形结构的面向对象实现提供了一种灵活的解决方案，通过叶子节点和树枝节点的递归组合，可以形成复杂的树形结构，但对树形结构的控制却非常简单
- **享元模式 Flyweight Pattern**
  - 享元模式主要用于减少创建对象的数量，以减少内存占用和提高性能, 运用共享技术有效地支持大量细粒度的对象
  - 它通过共享已经存在的对象来大幅度减少需要创建的对象数量、避免大量相似对象的开销，从而提高系统资源的利用率
  - 享元 (Flyweight) 模式中存在以下两种状态:
    - 内部状态, 即不会随着环境的改变而改变的可共享部分
    - 外部状态, 指随环境改变而改变的不可以共享的部分, 享元模式的实现要领就是区分应用中的这两种状态, 并将外部状态外部化
  - 享元模式的主要有以下角色:
    - 抽象享元角色 (Flyweight): 通常是一个接口或抽象类，在抽象享元类中声明了具体享元类公共的方法，这些方法可以向外界提供享元对象的内部数据（内部状态），同时也可以通过这些方法来设置外部数据（外部状态）

        - 具体享元角色 (Concrete Flyweight): 它实现了抽象享元类，称为享元对象；在具体享元类中为内部状态提供了存储空间。通常我们可以结合单例模式来设计具体享元类，为每一个具体享元类提供唯一的享元对象
        - 非享元角色 (Unsharable Flyweight): 并不是所有的抽象享元类的子类都需要被共享，不能被共享的子类可设计为非共享具体享元类；当需要一个非共享具体享元类的对象时可以直接通过实例化创建
        - 享元工厂角色 (Flyweight Factory): 负责创建和管理享元角色。当客户对象请求一个享元对象时，享元工厂检查系统中是否存在符合要求的享元对象，如果存在则提供给客户；如果不存在的话，则创建一个新的享元对象
    - 优点
        - 极大减少内存中相似或相同对象数量，节约系统资源，提供系统性能
        - 享元模式中的外部状态相对独立，且不影响内部状态
    - 缺点
        - 为了使对象可以共享，需要将享元对象的部分状态外部化，分离内部状态和外部状态，使程序逻辑复杂

5.7 行为型模式

- **行为型模式 (Behavioral Patterns)**
    - 行为型模式用于描述程序在运行时复杂的流程控制，即描述多个类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，它涉及算法与对象间职责的分配
    - 行为型模式分为类行为模式和对象行为模式，前者采用继承机制来在类间分派行为，后者采用组合或聚合在对象间分配行为
    - 由于组合关系或聚合关系比继承关系耦合度低，满足“合成复用原则”，所以对象行为模式比类行为模式具有更大的灵活性
    - **模板方法模式**和**解释器模式**是类行为型模式
- **模板方法模式 Template Method Pattern**
    - 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。模板方法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
    - 模板方法 (Template Method) 模式包含以下主要角色:
        - 抽象类 (Abstract Class): 负责给出一个算法的轮廓和骨架, 它由一个模板方法和若干个基本方法构成
            - 模板方法: 定义了算法的骨架，按某种顺序调用其包含的基本方法
            - 基本方法: 是实现算法各个步骤的方法，是模板方法的组成部分。基本方法又可以分为三种:
                - 抽象方法(Abstract Method): 一个抽象方法由抽象类声明、由其具体子类实现
                - 具体方法(Concrete Method): 一个具体方法由一个抽象类或具体类声明并实现，子类可以进行覆盖也可以直接继承
                - 钩子方法(Hook Method): 在抽象类中已经实现，包括用于判断的逻辑方法和需要子类重写的空方法两种

一般钩子方法是用于判断的逻辑方法，这类方法名一般为isXxx，返回值类型为boolean类型

        - 具体子类 (Concrete Class): 实现抽象类中所定义的抽象方法和钩子方法，它们是一个顶级逻辑的组成步骤
    - 优点
        - 提高代码复用性, 将相同部分的代码放在抽象的父类中，而将不同的代码放入不同的子类中
        - 实现了反向控制, 通过父类调用子类的操作，通过子类的具体实现扩展不同的行为，实现了反向控制
    - 缺点
        - 对每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象
        - 父类中的抽象方法由子类实现，子类执行的结果会影响父类的结果，这导致一种反向的控制结构，它提高了代码阅读的难度
- **策略模式 Strategy Pattern**
    - 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
    - 策略模式创建表示各种**策略对象**和一个行为随着策略对象改变而改变的 **context 对象**, 策略对象改变 context 对象的执行算法
    - 策略模式的主要角色如下:
        - 抽象策略 (Strategy) 类: 这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口
        - 具体策略 (Concrete Strategy) 类: 实现了抽象策略定义的接口，提供具体的算法实现或行为
        - 环境 (Context) 类: 持有一个策略类的引用，最终给客户端调用
    - 优点:
        - 策略类都实现同一个接口，所以使它们之间**可以自由切换**
        - 易于扩展, 增加一个新的策略只需要添加一个具体的策略类即可，基本不需要改变原有的代码，**符合"开闭原则"**
        - **避免使用多重条件选择语句** (if else), 充分体现面向对象设计思想
    - 缺点:
        - 客户端必须知道所有的策略类，并自行决定使用哪一个策略类
        - 策略模式将造成产生很多策略类，可以通过使用享元模式在一定程度上减少对象的数量
    - JDK源码 `Comparator` 使用了策略模式
        - 在Arrays类中有一个 `sort()` 方法, Arrays就是一个环境角色类，这个sort方法可以传一个新策略让Arrays根据这个策略来进行排序
        - 在调用Arrays的sort方法时，第二个参数传递的是Comparator接口的子实现类对象。所以Comparator充当的是抽象策略角色，而具体的子实现类充当的是具体策略角色
- **命令模式 Command Pattern**
    - 将一个请求封装为一个对象，使**发出请求的责任**和**执行请求的责任**分割开，这样两者之间通过**命令对象**进行沟通，方便将命令对象进行存储、传递、调用、增加与管理

- 命令模式包含以下主要角色:
  - 抽象命令类 (Command) 角色： 定义命令的接口，声明执行的方法
  - 具体命令 (Concrete Command) 角色: 具体的命令，实现命令接口；通常会持有接收者，并调用接收者的功能来完成命令要执行的操作
  - 实现者/接收者 (Receiver) 角色: 接收者，真正执行命令的对象。任何类都可能成为一个接收者，只要它能够实现命令要求实现的相应功能
  - 调用者/请求者 (Invoker) 角色: 要求命令对象执行请求，通常会持有命令对象，可以持有很多的命令对象。这个是客户端真正触发命令并要求命令执行相应操作的地方，也就是说相当于使用命令对象的入口
- 优点
  - 降低系统的耦合度。命令模式能将调用操作的对象与实现该操作的对象解耦
  - 增加或删除命令非常方便。采用命令模式增加与删除命令不会影响其他类，它满足“开闭原则”，对扩展比较灵活
  - 可以实现宏命令。命令模式可以与组合模式结合，将多个命令装配成一个组合命令，即宏命令
  - 方便实现 Undo 和 Redo 操作。命令模式可以与后面介绍的备忘录模式结合，实现命令的撤销与恢复
- 缺点
  - 使用命令模式可能会导致某些系统有过多的具体命令类
  - 系统结构更加复杂
- JDK源码 `Runable` 是一个典型命令模式
  - Runnable担当命令的角色，Thread充当的是调用者，start方法就是其执行方法
  -

- **职责链模式 Chain of Responsibility Pattern**
  - 为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止
  - 职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无须关心请求的处理细节和请求的传递，所以职责链将请求的发送者和请求的处理者解耦了
  - 职责链模式主要包含以下角色:
    - 抽象处理者 (Handler) 角色: 定义一个处理请求的接口，包含抽象处理方法和一个后继连接
    - 具体处理者 (Concrete Handler) 角色: 实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者
    - 客户类 (Client) 角色: 创建处理链，并向链头的具体处理者对象提交请求，它不关心处理细节和请求的传递过程
  - 优点:
    - 降低了对象之间的耦合度, 该模式降低了**请求发送者**和**接收者**的耦合度
    - 增强了系统的可扩展性, 可以根据需要增加新的请求处理类，满足开闭原则

        - 增强了给对象指派职责的灵活性, 当工作流程发生变化，可以动态地改变, 新增或者删除链内的成员或者修改它们的次序
        - 责任链简化对象之间的连接, 一个对象只需指向其后继者，不需保持其他所有处理者的引用，避免了巨大条件语句块
        - 责任分担, 每个类只需要处理自己的工作，不能处理的传递给下一个对象完成，明确各类的责任范围，符合类的单一职责原则
    - 缺点:
        - 不能保证每个请求一定被处理。由于一个请求没有明确的接收者，所以不能保证它一定会被处理，该请求可能一直传到链的末端都得不到处理
        - 对比较长的职责链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响
        - 职责链建立的合理性要靠客户端来保证，增加了客户端的复杂性，可能会由于职责链的错误设置而导致系统出错，如可能会造成循环调用
    - 在javaWeb应用开发中，FilterChain是职责链（过滤器）模式的典型应用
- **状态模式 State Pattern**
    - 类的行为是基于它的状态改变的, 我们创建表示各种状态的对象和一个行为随着状态对象改变而改变的 context 对象
    - 允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类
    - 状态模式包含以下主要角色:
        - 环境 (Context) 角色: 也称为上下文, 它定义了客户程序需要的接口, 维护一个当前状态, 并将与状态相关的操作委托给当前状态对象来处理
        - 抽象状态 (State) 角色: 定义一个接口，用以封装环境对象中的特定状态所对应的行为
        - 具体状态 (Concrete State) 角色: 实现抽象状态所对应的行为
    - 优点
        - 将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为
        - 允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块
    - 缺点
        - 状态模式的使用必然会增加系统类和对象的个数
        - 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱
        - 状态模式对"开闭原则"的支持并不太好
- **观察者模式 Observer Pattern**
    - 当对象间存在一对多关系时，则使用观察者模式。比如当一个对象被修改时，则会自动通知依赖它的对象
    - 观察者模式又被称为发布-订阅（Publish/Subscribe）模式，它定义对象间的一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态变化时，会通知所有的观察者对象 (依赖于它的对象)，使他们能够自动更新自己
    - 在观察者模式中有如下角色:
        - 抽象主题 (Subject 抽象被观察者): 抽象主题角色把所有观察者对象保存在一个集合里，每个主题都可以有任意数量的观察者，抽象主题提供一个接口，可以增加和删除观察者对象

        - 具体主题 (Concrete Subject 具体被观察者): 该角色将有关状态存入具体观察者对象，在具体主题的内部状态发生改变时，给所有注册过的观察者发送通知
        - 抽象观察者 (Observer): 是观察者的抽象类，它定义了一个更新接口，使得在得到主题更改通知时更新自己
        - 具体观察者 (Concrete Observer): 实现抽象观察者定义的更新接口，以便在得到主题更改通知时更新自身的状态
    - 优点
        - 降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。
        - 被观察者发送通知，所有注册的观察者都会收到信息【可以实现广播机制】
    - 缺点
        - 如果观察者非常多的话，那么所有的观察者收到被观察者发送的通知会耗时
        - 如果被观察者有循环依赖的话，那么被观察者发送通知会使观察者循环调用，会导致系统崩溃
    - 在 Java 中 java.util.Observable 类和 java.util.Observer 接口定义了观察者模式
        - 实现它们的子类就可以编写观察者模式实例
        - Observable 类是抽象目标类（被观察者），它有一个 Vector 集合成员变量，用于保存所有要通知的观察者对象
        - Observer 接口是抽象观察者，它监视目标对象的变化，当目标对象发生变化时，观察者得到通知，并调用 update 方法
- **中介者模式 Mediator Pattern**
    - 中介者模式是用来降低多个对象和类之间的通信复杂性。这种模式提供了一个中介类，通常处理不同类之间的通信，并支持松耦合，使代码易于维护
    - 用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，关系变为星型结构；而且可以独立地改变它们之间的交互，任何一个类的变动，只会影响的类本身以及中介者，减小系统的耦合
    - 中介者模式包含以下主要角色:
        - 抽象中介者 (Mediator) 角色: 它是中介者的接口，提供了同事对象注册与转发同事对象信息的抽象方法
        - 具体中介者 (Concrete Mediator) 角色: 实现中介者接口，定义一个 List 来管理同事对象，协调各个同事角色之间的交互关系，因此它依赖于同事角色
        - 抽象同事类 (Colleague) 角色: 定义同事类的接口，保存中介者对象，提供同事对象交互的抽象方法，实现所有相互影响的同事类的公共功能
        - 具体同事类 (Concrete Colleague) 角色: 是抽象同事类的实现者，当需要与其他同事对象交互时，由中介者对象负责后续的交互
    - 优点
        - 松散耦合
        - 集中控制交互
        - 一对多关联转变为一对一的关联
    - 缺点

        - 当对象类太多时，中介者的职责将很大，它会变得复杂而庞大，以至于系统难以维护
- **迭代器模式 Iterator Pattern**
    - 迭代器模式提供一种方法**顺序访问**一个聚合对象中各个元素, 而又无须暴露该对象的内部表示
    - 迭代器模式主要包含以下角色：
        - 抽象聚合 (Aggregate) 角色：定义存储、添加、删除聚合元素以及创建迭代器对象的接口
        - 具体聚合 (Concrete Aggregate) 角色: 实现抽象聚合类，返回一个具体迭代器的实例
        - 抽象迭代器 (Iterator) 角色: 定义访问和遍历聚合元素的接口，通常包含 hasNext()、next() 等方法
        - 具体迭代器 (Concretelterator) 角色: 实现抽象迭代器接口中所定义的方法，完成对聚合对象的遍历，记录遍历的当前位置
    - 优点
        - 它支持以不同的方式遍历一个聚合对象，在同一个聚合对象上可以定义多种遍历方式。在迭代器模式中只需要用一个不同的迭代器来替换原有迭代器即可改变遍历算法，我们也可以自己定义迭代器的子类以支持新的遍历方式
        - 迭代器简化了聚合类。由于引入了迭代器，在原有的聚合对象中不需要再自行提供数据遍历等方法，简化聚合类的设计
        - 在迭代器模式中，由于引入了抽象层，增加新的聚合类和迭代器类都很方便，无须修改原有代码，满足 “开闭原则” 的要求
    - 缺点
        - 增加了类的个数，这在一定程度上增加了系统的复杂性
- **访问者模式 Visitor Pattern**
    - 封装一些作用于某种数据结构中的各元素的操作，它可以在不改变这个数据结构的前提下定义作用于这些元素的新的操作
    - 主要将数据结构与数据操作分离
    - 访问者模式包含以下主要角色:
        - 抽象访问者 (Visitor) 角色：定义了对每一个元素（`Element`）访问的行为，它的参数就是可以访问的元素，它的方法个数理论上来讲与元素类个数（Element的实现类个数）是一样的，从这点不难看出，访问者模式要求元素类的个数不能改变
        - 具体访问者 (Concrete Visitor) 角色：给出对每一个元素类访问时所产生的具体行为
        - 抽象元素 (Element) 角色：定义了一个接受访问者的方法（`accept`），其意义是指，每一个元素都要可以被访问者访问
        - 具体元素 (Concrete Element) 角色： 提供接受访问方法的具体实现，而这个具体的实现，通常情况下是使用访问者提供的访问该元素类的方法
        - 对象结构 (Object Structure) 角色：定义当中所提到的对象结构，对象结构是一个抽象表述，具体点可以理解为一个具有容器性质或者复合对象特性的类，它会含有一组元素（`Element`），并且可以迭代这些元素，供访问者访问
    - 优点
        - 扩展性好, 在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能
        - 复用性好, 通过访问者来定义整个对象结构通用的功能，从而提高复用程度

        - 通过访问者分离无关行为, 把相关的行为封装在一起，构成一个访问者，这样每一个访问者的功能都比较单一
    - 缺点
        - 违背了开闭原则, 对象结构变化很困难, 在访问者模式中，每增加一个新的元素类，都要在每一个具体访问者类中增加相应的具体操作
        - 违反了依赖倒置原则, 访问者模式依赖了具体类，而没有依赖抽象类
    - 访问者模式用到了一种双分派的技术
        - 分派: 变量被声明时的类型叫做变量的**静态类型**，有些人又把静态类型叫做明显类型；而变量所引用的对象的**真实类型**又叫做变量的实际类型。比如 `Map map = new HashMap()` ，map 变量的静态类型是 `Map` ，实际类型是 `HashMap` 。根据对象的类型而对方法进行的选择，就是分派(Dispatch)，分派(Dispatch)又分为两种，即静态分派和动态分派
        - **静态分派(Static Dispatch)** 发生在编译时期，分派根据静态类型信息发生。方法重载就是静态分派
        - **动态分派(Dynamic Dispatch)** 发生在运行时期，动态分派动态地置换掉某个方法。Java通过方法的重写支持动态分派
    - 静态分派
        - 通过方法重载支持静态分派，重载方法的分派是根据静态类型进行的，这个分派过程在编译时期就完成了
    - 动态分派
        - 通过方法的重写支持动态分派，编译看左边，运行看右边
    - 双分派
        - 双分派技术就是在选择一个方法的时候，不仅仅要根据消息接收者（receiver）的运行时区别，还要根据参数的运行时区别
        - 既有方法重写 也有方法重载
        - 双分派实现动态绑定的本质，就是在重载方法委派的前面加上了继承体系中覆盖的环节，由于覆盖是动态的，所以重载就是动态的了
- **备忘录模式 Memento Pattern**
    - 备忘录模式提供了一种状态恢复的实现机制，使得用户可以方便地回到一个特定的历史步骤，当新的状态无效或者存在问题时，可以使用暂时存储起来的备忘录将状态复原，很多软件都提供了撤销（Undo）操作
    - 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态, 以便以后将对象恢复到原先保存的状态
    - 备忘录模式的主要角色如下:
        - 发起人 (Originator) 角色: 记录当前时刻的内部状态信息，提供创建备忘录和恢复备忘录数据的功能，实现其他业务功能，它可以访问备忘录里的所有信息
        - 备忘录 (Memento) 角色: 负责存储发起人的内部状态，在需要的时候提供这些内部状态给发起人
        - 管理者 (Caretaker) 角色: 对备忘录进行管理，提供保存与获取备忘录的功能，但其不能对备忘录的内容进行访问与修改

- 备忘录有两个等效的接口
  - **窄接口**：管理者(Caretaker)对象（和其他发起人对象之外的任何对象）看到的是备忘录的窄接口(narror Interface)，这个窄接口只允许他把备忘录对象传给其他的对象
  - **宽接口**：与管理者看到的窄接口相反，发起人对象可以看到一个宽接口(wide Interface)，这个宽接口允许它读取所有的数据，以便根据这些数据恢复这个发起人对象的内部状态
- 实现备忘录模式, 有两种方式:
  - “白箱”备忘录模式
  - “黑箱”备忘录模式
- “白箱”备忘录模式
  - 备忘录角色对任何对象都提供一个接口，即宽接口，备忘录角色的内部所存储的状态就对所有对象公开
  - 白箱备忘录模式是破坏封装性的。但是通过程序员自律，同样可以在一定程度上实现模式的大部分用意
- “黑箱”备忘录模式
  - 备忘录角色对发起人对象提供一个宽接口，而为其他对象提供一个窄接口
  - 在Java语言中，实现双重接口的办法就是将**备忘录类**设计成**发起人类**的内部成员类
- 优点:
  - 提供了一种可以恢复状态的机制。当用户需要时能够比较方便地将数据恢复到某个历史的状态
  - 实现了内部状态的封装。除了创建它的发起人之外，其他对象都不能够访问这些状态信息
  - 简化了发起人类。发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合单一职责原则
- 缺点: 资源消耗大。如果要保存的内部状态信息过多或者特别频繁，将会占用比较大的内存资源

- **解释器模式 Interpreter Pattern**
  - 给定一个语言，定义它的文法表示，并定义一个解释器，这个解释器使用该标识来解释语言中的句子
  - 抽象语法树
    - 在计算机科学中，抽象语法树（AbstractSyntaxTree，AST），或简称语法树（Syntax tree），是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构
- 解释器模式包含以下主要角色
  - 抽象表达式 (Abstract Expression) 角色：定义解释器的接口，约定解释器的解释操作，主要包含解释方法 interpret()。
  - 终结符表达式 (Terminal Expression) 角色：是抽象表达式的子类，用来实现文法中与终结符相关的操作，文法中的每一个终结符都有一个具体终结表达式与之相对应
  - 非终结符表达式 (Nonterminal Expression) 角色：也是抽象表达式的子类，用来实现文法中与非终结符相关的操作，文法中的每条规则都对应于一个非终结符表达式
  - 环境 (Context) 角色：通常包含各个解释器需要的数据或是公共的功能，一般用来传递被所有解释器共享的数据，后面的解释器可以从这里获取这些值

    - 客户端 (Client): 主要任务是将需要分析的句子或表达式转换成使用解释器对象描述的抽象语法树，然后调用解释器的解释方法，当然也可以通过环境角色间接访问解释器的解释方法
- 优点
    - 易于改变和扩展文法
    - 实现文法较为容易, 在抽象语法树中每一个表达式节点类的实现方式都是相似的，这些类的代码编写都不会特别复杂
    - 增加新的解释表达式较为方便, 如果用户需要增加新的解释表达式只需要对应增加一个新的终结符表达式或非终结符表达式类，原有表达式类代码无须修改，符合 "开闭原则"
- 缺点
    - 对于复杂文法难以维护, 如果一个语言包含太多文法规则，类的个数将会急剧增加，导致系统难以管理和维护
    - 执行效率较低, 在解释器模式中使用了大量的循环和递归调用，因此在解释复杂的句子时其速度很慢，而且代码调试也麻烦

Reference

- https://pdai.tech/
- https://blog.csdn.net/w20001118/article/details/125724647
- https://zhuanlan.zhihu.com/p/34426768
- https://www.liaoxuefeng.com/wiki/1252599548343744/1306581155184674
- volatile
- https://www.javatpoint.com/design-patterns-in-java
- https://www.runoob.com/design-pattern/singleton-pattern.html
- https://sourcemaking.com/design_patterns
- https://www.interviewbit.com/java-interview-questions/
- https://www.javatpoint.com/corejava-interview-questions
- Java8新特性

7. Java 基础

001. What make Java Write Once and Run Anywhere

- **The bytecode**
- Java compiler converts the Java programs into the class file (Byte Code) which is the intermediate language between source code and machine code. This bytecode is not platform specific and can be executed on any computer.

002. Java 5 Design Principles

- Java five Design Principles also called **SOLID Principles**
- **S**ingle Responsibility Principle (SRP)
- **O**pen for extension – Closed for modification Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependence Inversion Principle (DIP)

003. Access modifiers in Java

- Access modifiers, or access specifiers, The keywords which are used to define the access scope of the method, class, or a variable.
- **Public**: can be accessed by any class or method.
- **Protected**: can be accessed by the class of the same package, or by the sub-class of this class, or within the same class.
- **Default**: can be accessed within the package only.
- **Private**: can be accessed within the class only.

004. Purpose of static

- The methods or variables defined as static are **shared among all the objects of the class**.
- The static is the part of the class and not of the object. The static variables are **stored in the class area**, and **no need to create the object to access** such variables.
- Static is used where we need to define variables or methods which are **common to all the objects of the class**.

005. Purpose of final

- In Java, the final variable is used to restrict the user from updating it.
- **Stop value change, mathod overriding, inheritance**
- 修饰变量：表示变量一旦被赋值就不可以更改它的值
- 修饰方法：表示方法不可被子类重写，但是可以重载
- 修饰类：表示类不可被继承
- 修饰成员变量：
    - 如果final修饰的是类变量，只能在静态初始化块中指定初始值或者声明该类变量时指定初始值
    - 如果final修饰的是成员变量，可以在非静态初始化块、声明该变量或者构造器中执行初始值
- 修饰局部变量：
    - 系统不会为局部变量进行初始化，局部变量必须由程序员显示初始化
    - 因此使用final修饰局部变量时，即可以在定义时指定默认值（后面的代码不能对变量再赋值），也可以不指定默认值，而在后面的代码中对final变量赋初值（仅一次）

- The constructor can never be declared as final， compiler will throw an error
- An interface cannot be declare as final because the interface must be implemented by some class to provide its definition

006. What is the constructor

- The constructor can be defined as the special type of method that is **used to initialize the state of an object**.
- It is **invoked when the class is instantiated**, and the memory is allocated for the object.
- Every time, an object is created using the **new** keyword, the default constructor of the class is called.
- The name of the constructor must be similar to the class name.
- The constructor must not have an explicit return type, is not inherited, can't be final.
- **Types of constructors:**
    - Default Constructor
    - Parameterized Constructor

007. 重载和重写的区别

- 重写(Method Override)
    - Method Override is a subclass provides a specific implementation of a method which already provided by its parent class.
    - It is used for runtime polymorphism and to implement the interface methods.
    - **Must have same name, same signature**
    - 发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为private则子类就不能重写该方法
    - Cannot override the static method because they are the part of the class, not the object.
- 重载(Method Overload)
    - Method overloading is the polymorphism technique.
    - Allows to create multiple methods with the same name but different signature.
    - 发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同， 发生在编译时
- Cannot override or overload a private method, because its scope is limited to the class

008. 多态的理解

- Two types: **compile-time polymorphism** and **runtime polymorphism**
- Compile-time polymorphism
    - In compile-time polymorphism, call to a method is resolved at compile-time.
    - It is also known as **static binding**, early binding, or overloading.

    - **Overloading is a way to achieve compile-time polymorphism** in which, we can define multiple methods or constructors with different signatures.
    - It provides fast execution because the type of an object is determined at compile-time.
    - Compile-time polymorphism provides less flexibility because all the things are resolved at compile-time.
- Runtime polymorphism
    - In runtime polymorphism, call to an overridden method is resolved at runtime.
    - It is also known as **dynamic binding**, late binding, overriding, or dynamic method dispatch.
    - **Overriding is a way to achieve runtime polymorphism** in which, we can redefine some particular method or variable in the derived class. By using overriding, we can give some specific implementation to the base class properties in the derived class.
    - It provides slower execution as compare to compile-time because the type of an object is determined at run-time.
    - Run-time polymorphism provides more flexibility because all the things are resolved at runtime.

009. & 和 && 区别

- &运算符有两种用法：(1)位运算符；(2)逻辑运算符
- & 不管第一个表达式是否为真都会执行两个表达式
- &&运算符是短路运算符。逻辑与跟短路与的差别非常巨大，虽然二者都要求运算符左右两端的布尔值都是true 整个表达式的值才是 true。&&之所以称为短路运算，是因为如果&&左边的表达式的值是false，右边的表达式会被直接短路掉，不会进行运算
- 逻辑或运算符（|）和短路或运算符（||）的差别也是如此, 不做赘述

010. == 和 equals 区别

- ==是运算符，equals是方法
- ==用来比较变量的值是否相同，基本数据类型是比较变量值，引用类型是比较堆中内存对象的地址
- equals作为object中的方法，默认也是采用==比较，通常会重写比较的是对象的内容
- All in all, ==比较的是变量的值，equals比较的是对象的内容

011. hashCode和equals 区别

- hashCode()介绍
    - hashCode() 默认实现通过将对象的内存地址转换为整数来计算对象的哈希码，也称为散列码，返回一个int整数
    - 这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在JDK的Object.java中，这就意味着Java中的任何类都包含有hashCode()函数
- 说明为什么要有 hashCode

    - 当把对象加入HashSet 时，HashSet 会先计算对象的 hashcode 值来判断对象加入的位置，同时也会与其他已经加入的对象的 hashcode 值作比较，如果没有相符的hashcode，HashSet会假设对象没有重复出现
    - 但是如果发现有相同 hashcode 值的对象，这时会调用 equals()方法来检查 hashcode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置
    - 这样就大大减少了 equals 的次数，相应就大大提高了执行速度
- hashCode()与equals()的原则
    - 如果两个对象equals()相等，则hashcode一定也是相同的
    - 如果两个对象equals()不相等，hashCode() 有可能相等，也有可能不等
    - 如果两个对象hashCode()不相等，那两个对象equals()一定不相等

012. What is the abstraction

- Abstraction is a process of hiding the implementation details and showing only functionality to the user. It displays just the essential things to the user and hides the internal information
- Abstraction enables you to **focus on what the object does instead of how it does it**. Abstraction lets you focus on what the object does instead of how it does it.
- Two ways to achieve the abstraction: **Abstract Class** and **Interface**
- **Abstract Class**
    - **A class that is declared as abstract** is known as an abstract class.
    - It needs to be extended and its method implemented. It cannot be instantiated. It can have abstract methods, non-abstract methods, constructors, and static methods. It can also have the final methods which will force the subclass not to change the body of the method.
    - Abstract method must in abstract class
- **Interface**
    - **The interface is a blueprint for a class that has static constants and abstract methods.**
    - It can be used to achieve full abstraction and multiple inheritance.
    - There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
    - Java Interface also represents the IS-A relationship. It cannot be instantiated just like the abstract class. However, we need to implement it to define its methods.

013. 抽象类和接口的区别

- 从语法层面上看
    - 抽象类可以提供成员方法的实现细节，而接口中只能存在public abstract 方法，分号结尾，不带花括号
    - 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是public static final类型的
    - 抽象类可以有静态代码块和静态方法, 接口能含有静态代码块以及静态方法
    - 抽象类可以有构造函数, 而接口没有

    - 一个类只能继承一个抽象类，而一个类却可以实现多个接口
- 从设计层面上看
    - 抽象类是对一种事物的抽象，即对类抽象，而接口是对行为的抽象
    - 抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象
    - 抽象类作为很多子类的父类，它是一种模板式设计。而接口是一种行为规范，它是一种辐射式设计
- 从使用上来看
    - 一个类只能**继承一个抽象类**，而一个类却可以**实现多个接口**
    - 继承抽象类为**extends**，实现接口为**implements**
- 相同点
    - 接口和抽象类都不能被实例化
    - 实现接口或继承抽象类的普通子类都必须实现这些抽象方法后才能实例化
- 在很多情况下，接口优先于抽象类，因为接口没有抽象类严格的类层次结构要求，可以灵活地为一个类添加行为
- 当你关注一个事物的本质的时候，用抽象类；当你关注一个操作的时候，用接口

014. Java异常体系

- Exception Handling is a mechanism that is used to handle runtime errors.
    - It is used primarily to handle checked exceptions.
- Java中的所有异常都来自顶级父类 **Throwable**
- Throwable下有两个子类 **Error** 和 **Exception**
- Error 是程序无法处理的错误，一旦出现这个错误，则程序将被迫停止运行
    - OutOfMemoryError, AssertionError
- Exception不会导致程序停止，分为两个部分CheckedException检查异常 和 Unchecked Exception运行时异常
- CheckedException常常发生在程序编译过程中，会导致程序编译不通过，**可以被捕获**
    - SQLException, ClassNotFoundException, IOException
- RunTimeException常常发生在程序运行过程中，会导致程序当前线程执行失败
    - ArithmaticException, NullPointerException

015. What is String Pool

- String pool is the space reserved in the heap memory that can be used to store the strings.
- The main advantage of using the String pool is whenever we create a string literal; the JVM checks the "string constant pool" first.
    - If the string already exists in the pool, a reference to the pooled instance is returned.
    - If the string doesn't exist in the pool, a new string instance is created and placed in the pool.
- Therefore, it **saves the memory** by avoiding the duplicacy.

- Java uses the concept of the string literal
  - Suppose there are five reference variables, all refer to one object "sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables.
  - That is why string objects are **immutable** in java.

016. Two ways create String

- String Literal:
  - Java String literal is created by using double quotes, placed in the string constant pool
- By new keyword:
  - JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the constant string pool. The variable s will refer to the object in a heap (non-pool).

017. String, StringBuffer, StringBuilder

- 可变性：String 不可变，StringBuffer 和 StringBuilder 可变
- String 是final不可变的，对String每次改变都会新生成一个String对象，然后把指针指向新的引用对象
- 线程安全 Thread safety
  - String 不可变，因此是线程安全的
  - StringBuilder 不是线程安全的
  - StringBuffer 是线程安全的，内部使用 synchronized 进行同步
- 使用的总结
  - 如果要操作少量的数据用String
  - 单线程操作字符串缓冲区 下操作大量数据用StringBuilder
  - 多线程操作字符串缓冲区 下操作大量数据用StringBuffer

018. Java Serialization

- **Serialization** in Java is a mechanism of writing the state of an object into a byte stream.
- It is mainly used to travel object's state on the network (known as marshaling).
- Serializable interface is used to perform serialization.

```
// class object e implements Serializable
FileOutputStream fileOut = new FileOutputStream("./file.ser");
ObjectOutputStream out = new ObjectOutputStream(fileOut);
out.writeObject(e);
out.close();
fileOut.close();
```

- The content can be restored using **deserialization**.

```
// class object e implements Serializable
FileInputStream fileIn = new FileInputStream(("./file.ser");
ObjectInputStream in = new ObjectInputStream(fileIn);
e = (e转型) in.readObject();
in.close();
fileIn.close();
```

019. Java Reflection

- Reflection is the process of examining or modifying the runtime behavior of a class at runtime.
- The java.lang.Class class provides various methods that can be used to get metadata, examine and change the runtime behavior of a class. The java.lang and java.lang.reflect packages provide classes for java reflection.
- JVM为每个加载的`class`创建了对应的`Class`实例
  - 如果获取了某个`Class`实例，我们就可以通过这个`Class`实例获取到该实例对应的`class`的所有信息
- Three ways to get the Class's class instance
  - 直接通过一个`class`的静态变量`class`获取：the **.class** syntax

    ```
    Class cls = String.class;
    ```

  - 如果有一个实例变量：**getClass()** method of Object class

    ```
    String s = "Hello";
    Class cls = s.getClass();
    ```

  - 如果知道一个`class`的完整类名: **forName()** method of Class class

    ```
    Class cls = Class.forName("java.lang.String");
    ```

- 创建class 变量, 调用Class提供的newInstance()方法： (class type)class obj.newInstance();

```
Person p1 = new Person();
Person p2 = (Person) p1.class.newInstance();
```

020. Array 和 Collection 区别

- Array 和 Collection 在存储对象引用和操作数据方面相似
- 其主要区别：
  - Arrays are always of **fixed size**; but Collection's size can be **changed dynamically** as per need.
  - Arrays can only **store homogeneous** objects; but Collection can store **heterogeneous objects**.

    - Arrays cannot provide the ready-made methods for user requirements as sorting, searching; but Collection includes **readymade methods** to use..

Comparable 和 Comparator区别

- Comparable
    - Comparable是排序接口。若一个类实现了Comparable接口，就意味着该类支持排序
    - 需要实现接口方法`compareTo(T o1)`
    - 实现了Comparable接口的类的对象的列表或数组可以通过Collections.sort或Arrays.sort进行自动排序
- Comparator
    - Comparator是比较接口，我们如果需要控制某个类的次序，而该类本身不支持排序，那么可以实现Comparator接口
    - 需要实现接口方法`compareTo(T o1, T o2)`，但可以不实现`equals(Object obj)`函数
    - compare函数返回“负数”，意味着“o1比o2小”；返回“零”，意味着“o1等于o2”；返回“正数”，意味着“o1大于o2”
- 用Comparable需要修改源代码, 用Comparator 不需要修改源代码， 而是另外实现一个比较器

021. ArrayList 和 Vector 区别

- **四点：synchronized，legacy，expansion，thread-safe**
- ArrayList is **not synchronized**; Vector is **synchronized**.
- ArrayList is not a legacy class; Vector is a legacy class.
- ArrayList increases its size by **50% of the array size**; Vector increases its size by **doubling the array size**.
- ArrayList is **not thread-safe** as it is not synchronized; Vector list is **thread-safe** as its every method is synchronized.

022. ArrayList 和 LinkedList 区别

- **五点：底层实现，操作效率，适合场景，能否随机访问，内存开销**
- 底层数据结构不同
    - ArrayList底层是**基于动态数组 dynamic array** 实现的，**访问元素速度优于 LinkedList**，默认容量为10，当元素数量到达容量时，生成一个新的数组，大小为前一次的1.5倍，然后将原来的数组copy过来
    - LinkedList底层是**基于双向链表 doubly linked list** 实现的，在内存中是离散的，没有扩容机制，**批量插入或删除数据时优于 ArrayList**
- 由于底层数据结构不同，他们所适用的场景也不同
    - ArrayList更适合**存储和随机查找**，数组在内存中是连续的地址，所以查找数据更快，但由于扩容机制添加数据效率更低
    - LinkedList更适合**删除和添加**，在查找数据时需要从头遍历，所以查找慢，但是添加数据效率更高
- 内存开销不同

    - ArrayList takes less memory overhead as it stores **only object**
    - LinkedList takes more memory overhead, as it stores the **object as well as the address** of that object.
- 另外ArrayList和LinkedList都实现了List接口，但是LinkedList还额外实现了Deque接口，所以LinkedList还可以当做队列来使用

023. Iterator 和 ListIterator 区别

- The Iterator traverses the elements in the forward direction only; ListIterator traverses the elements in backward and forward directions both.
- The Iterator can be used in List, Set, and Queue; ListIterator can be used in List only.
- The Iterator can only perform remove operation while traversing the collection; ListIterator can perform add, remove, and set operation while traversing the collection.

024. Iterator 和 Enumeration 区别

- The Iterator can traverse **legacy and non-legacy** elements； Enumeration can traverse only **legacy** elements.
- The Iterator is **fail-fast**; Enumeration is **not fail-fast**.
- The Iterator is **slower** than Enumeration; Enumeration is faster than Iterator.
- The Iterator can perform remove operation while traversing the collection; The Enumeration can perform **only traverse** operation on the collection.

025. 如何使ArrayList线程安全

- 使用Vector，Vector底层与Arraylist相同，但是每个方法都由synchronized修饰，速度很慢
- 使用collections.synchronizedList() 方法为ArrayList加锁

```
// creating an ArrayList object
List<String> list = new ArrayList<>();
// Synchronizing the list
List<String> synlist = Collections.synchronizedList(list);
```

- 使用 JUC 下的CopyOnWriterArrayList，该类实现了读操作不加锁，写操作时为list创建一个副本，期间其它线程读取的都是原本list，写操作都在副本中进行，写入完成后，再将指针指向副本

```
CopyOnWriteArrayList c = new CopyOnWriteArrayList();
// Creates a list containing the elements of the specified collection
CopyOnWriteArrayList c = new CopyOnWriteArrayList(Collection obj);
// Creates a list holding a copy of the given array.
CopyOnWriteArrayList c = new CopyOnWriteArrayList(Object[] obj);
```

026. List和Set的区别

- 三点：是否包含唯一元素，是否有序，是否允许多个Null
- List：
  - 有序，按对象进入的顺序保存对象，可重复，允许多个Null元素对象
  - 可以使用Iterator取出所有元素，在逐一遍历，还可以使用get(int index)获取指定下标的元素
- Set：
  - 无序，不可重复，最多允许有一个Null元素对象
  - 取元素时只能用Iterator接口取得所有元素，在逐一遍历各个元素

027. HashMap Jdk1.7到1.8 区别

- 1.7版本
  - 底层是数组+链表
  - 链表插入使用的是头插法，并发下头插法会造成死循环
- 1.8版本
  - 底层是数组+链表+红黑树
  - 加红黑树的目的是提高HashMap插入和查询整体效率
  - 链表插入使用的是尾插法，因为插入key和value时需要判断链表元素个数，所以需要遍历链表统计链表元素个数，所以正好就直接使用尾插法

028. HashMap的扩容机制

- HashMap的扩容机制：HashMap的默认容量为16，默认的负载因子(loadFactor) 为0.75，当HashMap中元素个数超过容量(capacity) 乘以负载因子时，就创建一个大小为前一次两倍的新数组，再将原来数组中的数据复制到新数组中
- 当数组长度到达64且链表长度大于8时，链表转为**红黑树**, 从树转为链表的阈值为6；这是为了防止树化、线化阈值相同而导致频繁转换, 如果用B+树的话，在数据量不是很多的情况下，数据都会“挤在”一个结点里面。这个时候遍历效率就退化成了链表
- 1.7版本
  - 首先生成数组，遍历老数组每一个元素，根据元素key和新数组长度决定元素的新位置
  - 所有元素转移完，将新数组赋值给hashmap的table属性
- 1.8版本
  - 首先生成数组，遍历老数组每一个位置上的链表或者红黑树
  - 如果是链表，直接添加到新位置
  - 如果是红黑树，先算出树中每个元素在新数组对应位置
    - 统计每个位置下标个数
    - 如果超过8，转变为红黑树；否则直接生成链表
  - 所有元素转移完，将新数组赋值给hashmap的table属性

029. HashMap的Put方法

- 根据Key通过哈希算法与与运算得出数组下标
- 如果数组下标位置元素为空，则将key和value封装为Entry对象并放入该位置
  - JDK1.7中是Entry对象，JDK1.8中是Node对象
- 如果数组下标位置元素不为空，则要分情况讨论
  - 如果是JDK1.7，则先判断是否需要扩容，如果要扩容就进行扩容，如果不用扩容就生成Entry对象，并使用头插法添加到当前位置的链表中
  - 如果是JDK1.8，则会先判断当前位置上的Node的类型，看是红黑树Node，还是链表Node
    - 如果是红黑树Node，则将key和value封装为一个红黑树节点并添加到红黑树中去，在这个过程中会判断红黑树中是否存在当前key，如果存在则更新value
    - 如果此位置上的Node对象是链表节点，则将key和value封装为一个链表Node并通过尾插法插入到链表的最后位置去，因为是尾插法，所以需要遍历链表，在遍历链表的过程中会判断是否存在当前key，如果存在则更新value，当遍历完链表后，将新链表Node插入到链表中，插入到链表后，会看当前链表的节点个数，如果大于等于8，那么则会将该链表转成红黑树
    - 将key和value封装为Node插入到链表或红黑树中后，再判断是否需要进行扩容，如果需要就扩容，如果不需要就结束PUT方法

030. HashMap长度为2的幂次方原因

- 为了能让 HashMap 存取高效，尽量把数据分配均匀
- Hash 值的范围值很大，大概40亿映射空间，但是散列值不能直接使用，要对数组长度取模运算，得到的余数是对应的数组下标
- 取余(%)操作等价于与其除数减一的与操作 (hash%length==hash&(length-1)，前提 length 是2的n次方
  - 二进制算法1000 - 1 = 0111, &时候高位全部舍弃，只保留了hash值最后n位
- 因为二进制位操作，hash&(length-1) 比 hash%length 运算更快

031. HashMap 线程不安全

- 多个线程对同个桶下标put操作可能覆盖
- HashMap的主干是一个Node 数组, Node 是HashMap的基本组成单元
- 每一个Node包含一个key-value键值对, Node 是HashMap中的一个静态内部类

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash; // 对key的hashcode值进行hash运算后得到的值，存储在Node，避免重复计算
    final K key;
    V value;
    Node<K,V> next; // 存储指向下一个Node的引用，单链表结构
}

transient int size; // 实际存储的key-value键值对的个数
int threshold; // threshold一般为 capacity*loadFactor
final float loadFactor; // 负载因子，代表了table的填充度有多少，默认是0.75
```

032. 如何使HashMap线程安全

- 使用HashTable

```
Hashtable<String, Integer> hashtable = new Hashtable<>();
```

- Collections.synchronizedHashMap()方法

```
// create a HashMap object
Map<String, String> map = new HashMap<String, String>();
// Synchronizing the map
Map<String, String> synmap = Collections.synchronizedMap(map);
```

- 使用ConcurrentHashMap:

```
ConcurrentHashMap<Integer, Integer> map = new ConcurrentHashMap<>();
```

033. HashMap 和 HashTable 区别

- **四点：synchronized，能否有null值，线程不安全，遍历方式**
- HashMap方法没有synchronized修饰；HashTable的每个方法都用synchronized修饰同步, 每次锁住整张HashTable让线程独占, 但同时读写效率很低
- HashMap线程不安全；HashTable 线程安全
- HashMap允许key和value为null；HashTable的Key不允许为null
- 遍历方式的内部实现不同，HashMap、Hashtable都支持 Iterator，但是Hashtable还支持 Enumeration

034. HashSet 和 TreeSet 区别

- **四点：HashSet无序TreeSet升序；基于哈希表/基于树实现；HashSet 更快；由HashMap/TreeMap 支持**
- HashSet maintains no order whereas TreeSet maintains ascending order.
- HashSet impended by hash table whereas TreeSet implemented by a Tree structure.
- HashSet performs faster than TreeSet.
- HashSet is backed by HashMap whereas TreeSet is backed by TreeMap.

035. HashMap 和 TreeMap 区别

- **四点：HashMap无序TreeMap升序；基于哈希表/基于树实现；允许值键/键排序；能否包含空值**
- HashMap maintains no order, but TreeMap maintains ascending order.
- HashMap is implemented by hash table whereas TreeMap is implemented by a Tree structure.
- HashMap can be sorted by Key or value whereas TreeMap can be sorted by Key.
- HashMap may contain a null key with multiple null values whereas TreeMap cannot hold a null key but can have multiple null values.

036. 泛型中extends和super的区别

- <? extends T>表示包括T在内的任何T的子类
- <? super T>表示包括T在内的任何T的父类

8. Java 并发

001. Advantages of Multithreading

- Multithreading allows an program to be always reactive for input, even already running with some background tasks
- Multithreading allows the faster execution of tasks, as threads execute independently.
- Multithreading provides better utilization of cache memory as threads share the common memory resources.
- Multithreading reduces the number of the required server as one server can execute multiple threads at a time.

002. 线程的生命周期

- 线程通常有五种状态，创建，就绪，运行、阻塞和死亡状态：
  - 新建状态(**New**): 新创建了一个线程对象, 但调用 start() 方法之前，线程不会启动
  - 就绪状态(**Runnable**): 线程调用 start() 方法后，线程位于可运行线程池中变得可运行，等待获取CPU的使用权
  - 运行状态 (**Running**): 线程调度器从就绪状态的线程中挑选线程，运行代码
  - 阻塞状态 (**Waiting/Blocked**): 线程因为某种原因放弃CPU使用权暂时停止运行，但仍然存活alive
  - 死亡状态 (**Dead/Terminated**): 线程执行完或因异常退出了run方法，该线程结束生命周期
- 阻塞的情况又分为三种
  - 等待阻塞：运行的线程执行wait方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify或notifyAll方法才能被唤醒，wait是object类的方法
  - 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“锁池”中
  - 其他阻塞：运行的线程执行sleep或join方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep状态超时、join等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。sleep是Thread类的方法

003. 并发 并行 串行区别

- 串行 (serial): 在时间上不可能发生重叠，前一个任务没结束，下一个任务就只能等着
- 并行 (parallel): 在时间上是重叠的，两个任务在同一时刻互不干扰的同时执行
- 并发 (concurrent): 允许两个任务彼此干扰。统一时间点、只有一个任务运行，交替执行

004. 并发的三大特性

- Atomicity, Visibility, Ordering
- 原子性 Atomicity
  - 原子性是指在一个操作中cpu不可以在中途暂停然后再调度，即**不被中断操作**，要不全部执行完成，要不都不执行
  - 比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。2个操作必须全部完成
  - 那程序中原子性指的是最小的操作单元，比如自增操作，它本身其实并不是原子性操作，分了3步的，包括读取变量的原始值、进行加1操作、写入工作内存。所以在多线程中，一个线程可能还没自增完，才执行到第二步，另一个线程就已经读取了值，导致结果错误。如果我们能保证自增操作是一个原子性的操作，那么就能保证其他线程读取到的一定是自增后的数据
  - 关键字：synchronized
- 可见性 Visibility
  - 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
  - 若两个线程在不同的cpu，那么线程1改变了i的值还没刷新到主存，线程2又使用了i，那么这个i值肯定还是之前的，线程1对变量的修改线程没看到这就是可见性问题
  - 关键字：volatile、synchronized、final
- 有序性 Ordering
  - 虚拟机在进行代码编译时，对于那些改变顺序之后不会对最终结果造成影响的代码，虚拟机不一定会按照我们写的代码的顺序来执行，有可能将他们重排序。实际上，对于有些代码进行重排序之后，虽然对变量的值没有造成影响，但有可能会出现线程安全问题
  - 关键字：volatile、synchronized
  - volatile本身就包含了禁止指令重排序的语义，synchronized关键字是由“一个变量在同一时刻只允许一条线程对其进行lock操作”这条规则明确的
- synchronized关键字同时满足以上三种特性，但是volatile关键字不满足原子性
  - 在某些情况下，volatile的同步机制的性能确实要优于锁(使用synchronized关键字或java.util.concurrent包里面的锁)，因为volatile的总开销要比锁低
  - 我们判断使用volatile还是加锁的唯一依据就是volatile的语义能否满足使用的场景(原子性)

005. Thread 和 Runable 区别

- Two ways to create Thread:
  - By extending the Thread class 继承Thread类
  - By implementing the Runnable interface 实现Runnable接口
- Extending the Thread class then cannot extend other class; implementing the Runnable interface can extend class
- Thread实现了Runnable接口并进行了扩展，而Thread和Runnable的实质是实现的关系，没有可比性
- 无论使用Runnable还是Thread，都会new Thread，然后执行run方法

- 用法上，如果有复杂的线程操作需求，那就选择继承Thread，如果只是简单的执行一个任务，那就实现runnable
- Thread class 提供 inbuilt methods such as getPriority(), isAlive(); Runnable interface 只提供 run().

006. 线程使用方式

- extend Thread class:
  - class继承Thread并实现 run() ，创建object并调用start() 方法
- implement Runnable interface:
  - class实现Runnable 接口并实现 run()，创建object，用Thread封装并调用 start()
- implement Callable interface:
  - class实现Callable 接口并实现 call()，创建object，用FutureTask封装，再用Thread封装并调用 start()
  - FutureTask object.get() 可以得到返回值
- 使用 threadpool()
- Thread cLass methods:
  - start(): start the thread; call run method for the thread
  - run()
  - currentThread()
  - getNAME(); setName();
  - yield(); join();
  - stop(): deprecated; destroy():deprecated;
  - sleep(long milli time)
  - isAlive()

007. sleep() wait() park()

sleep() 和 wait() 区别

- sleep()
  - sleep() method is used to block or pause the execution of a thread for a particular time.
  - It pauses the execution of the current thread for the given time and gives priority to another thread(if available).
  - When the waiting time completed then again previous thread changes its state from **waiting** to **runnable**
- wait()
  - Provided by the Object class in Java. This method is used for Java inter-thread communication.
  - wait() is used to pause the current thread, and wait until another thread call the notify() or notifyAll()

    - wait() method be called from the synchronized block
- sleep() 和 wait() 的区别
    - sleep 是 **Thread 类的静态方法**，wait 则是 **Object 类的方法**
    - sleep 方法**不会释放lock**，wait**会释放lock并加入到等待队列中**
        - sleep 就是把cpu的执行资格和执行权释放出去，不再运行此线程，当定时时间结束再取回cpu资源，参与cpu的调度，获取到cpu资源后就可以继续运行
        - 如果sleep时该线程有锁，那么sleep不会释放这个锁，而是把锁带着进入了冻结状态，其他需要这个锁的线程不可能获取到这个锁。如果在睡眠期间其他线程调用了这个线程的interrupt方法，那么这个线程也会抛出interruptexception异常返回，这点和wait是一样的
    - sleep 方法**不依赖于synchronized**，但是wait需要**依赖synchronized关键字** (在synchronized方法/代码块中执行)
    - sleep **自动唤醒** (休眠之后推出阻塞), 但是wait要**调用notify()或notifyall()唤醒**
    - sleep 一般用于**当前线程休眠**，或者轮询暂停操作，wait **多用于多线程之间的通信**
    - **sleep 会强制让出 CPU 执行时间且强制上下文切换，而 wait 则不一定，wait 后可能还是有机会重新竞争到锁继续执行的**

sleep() 和 park() 的区别

- sleep() 和 park()都是阻塞当前线程的执行，且都不会释放当前线程占有的锁资源
- sleep()没法从外部唤醒; LockSupport.park()方法可以被另一个线程调用LockSupport.unpark()方法唤醒
- sleep()方法声明上抛出了InterruptedException中断异常，调用者需要捕获这个异常；park()方法不需要捕获中断异常
- sleep()本身就是一个native方法；LockSupport.park()底层是调用的Unsafe的native方法

wait() 和 park() 的区别

- wait()方法需要在synchronized块中执行；park()可以在任意地方执行
- wait()方法声明抛出了中断异常，调用者需要捕获或者再抛出；park()不需要捕获中断异常
- wait()不带超时的，需要另一个线程执行notify()来唤醒，但不一定继续执行后续内容；park()不带超时的，需要另一个线程执行unpark()来唤醒，一定会继续执行后续内容

如果在wait()之前执行了notify()会怎样

- 如果当前的线程不是此对象锁的所有者，却调用该对象的notify()或wait()方法时抛出IllegalMonitorStateException异常
- 如果当前线程是此对象锁的所有者，wait()将一直阻塞，因为后续将没有其它notify()唤醒它

如果在park()之前执行了unpark()会怎样?

- 线程不会被阻塞，直接跳过park()，继续执行后续内容

008. join() 和 yield() 区别

- sleep()
  - Thread.sleep(millisec) 方法会休眠当前正在执行的线程，millisec 单位为毫秒
  - sleep() 可能会抛出 InterruptedException，因为异常不能跨线程传播回 main() 中，因此必须在本地进行处理
  - 线程中抛出的其它异常也同样需要在本地进行处理
- yield()
  - 对静态方法 Thread.yield() 的调用声明了当前线程已经完成了生命周期中最重要的部分，可以切换给其它线程来执行
  - 该方法只是对线程调度器的一个建议，而且也只是建议具有相同优先级的其它线程可以运行(推出线程然后所有重新争夺)
- join()
  - join() method causes the currently running threads to stop executing until the joined thread completes its task.
- join() 和 yield()的区别
  - yield() 执行后线程直接进入就绪状态，马上释放了cpu的执行权，但是依然保留了cpu的执行资格，所以有可能cpu下次进行线程调度还会让这个线程获取到执行权继续执行
  - join() 执行后线程进入阻塞状态，例如在线程A中调用线程B的join()，那线程A会进入到阻塞队列，直到线程B结束或中断线程

009. synchronization 理解

- Synchronization is the capability to control the access of multiple threads to any shared resource.
  - To prevent thread interference.
  - To prevent consistency problem.
- When the multiple threads try to do the same task, there is a possibility of an erroneous result, hence to remove this issue, Java uses the process of synchronization which allows only one thread to be executed at a time.
- Synchronization can be achieved in three ways:
  - by the synchronized method
  - by synchronized block
  - by static synchronization (lock a class)
- purpose of the Synchronized
  - Only one thread at a time can execute on a particular resource, and all other threads which attempt to enter the synchronized block are blocked.
  - Synchronized block is used to lock an object for any shared resource.

010. 线程之间的互斥

- 两种锁机制来控制多个线程对共享资源的互斥访问: JVM 实现的 **synchronized** 和 JDK 实现的 **ReentrantLock**
- **synchronized**
  - **同步一个代码块**: `synchronized (this) {...}`
  - 它只作用于同一个对象，如果调用两个对象上的同步代码块，就不会进行同步
  - **同步一个方法**: `public synchronized void func () {...}`
  - **同步一个类**: 作用于整个类，也就是说两个线程调用同一个类的不同对象上的这种同步语句，也会进行同步
- **ReentrantLock**
  - ReentrantLock 是 java.util.concurrent(J.U.C)包中的锁
- Synchronized和lock的区别
  - synchronized 是内置关键字，lock是一个类
  - synchronized 会自己加锁释放锁，只有代码执行完毕或者异常结束才会释放锁，lock需要手动加锁释放锁
  - synchronized 是JVM层次通过监视器实现的，lock 是通过AQS实现的
  - synchronized 无法判断获取锁的状态, lock 可以判断是否获取到了锁
  - synchronized 不能设定超时，不能中断一个正在使用锁的线程; lock 锁可以中断和设置超时
  - synchronized 是可重入锁、**非公平锁**、不可中断锁，lock的ReentrantLock是可重入锁，可中断锁，**可以是公平锁或非公平锁**
  - synchronized 适合锁少量的代码同步问题, lock 适合锁大量的同步代码

011. 线程之间的协作

- 当多个线程可以一起工作去解决某个问题时，如果某些部分必须在其它部分之前完成，那么就需要对线程进行协调
- join() 在线程中调用另一个线程的 join() 方法，会将当前线程挂起，而不是忙等待，直到目标线程结束
- wait() notify() notifyAll()
  - 调用 wait() 使得线程等待某个条件满足，线程在等待时会被挂起，当其他线程的运行使得这个条件满足时，其它线程会调用 notify() 或者 notifyAll() 来唤醒挂起的线程
  - 它们都属于 Object 的一部分，而不属于 Thread
  - 只能用在同步方法或者同步控制块中使用，否则会在运行时抛出 IllegalMonitorStateExeception
  - 使用 wait() 挂起期间，线程会释放锁。这是因为，如果没有释放锁，那么其它线程就无法进入对象的同步方法或者同步控制块中，那么就无法执行 notify() 或者 notifyAll() 来唤醒挂起的线程，造成死锁
- await() signal() signalAll()
  - java.util.concurrent 类库中提供了 Condition 类来实现线程之间的协调，可以在 Condition 上调用 await() 方法使线程等待，其它线程调用 signal() 或 signalAll() 方法唤醒等待的线程
  - 相比 wait() 这种等待方式，await() 可以指定等待的条件，因此更加灵活

    - Condition是个接口，基本的方法就是await()和signal()方法, 调用Condition的await()和signal()方法，都必须在lock保护之内，就是说必须在lock.lock() 和lock.unlock() 之间才可以使用
- 锁池和等待池
    - 在java中，每个对象都有两个池，锁池和等待池
    - 锁池：所有需要竞争同步锁的线程都会放在锁池当中，比如当前对象的锁已经被其中一个线程得到，则其他线程需要在这个锁池进行等待，当前面的线程释放同步锁后锁池中的线程去竞争同步锁，当某个线程得到后会进入就绪队列进行等待cpu资源分配
    - 等待池：当我们调用wait() 方法后，线程会放到等待池当中，等待池的线程是不会去竞争同步锁。只有调用了notify() 或notifyAll() 后等待池的线程才会开始去竞争锁，notify() 是随机从等待池选出一个线程放到锁池，而notifyAll() 是将等待池的所有线程放到锁池当中

012. 线程安全的理解

- 当多个线程访问一个对象时，如果不用进行额外的同步控制或其他的协调操作，调用这个对象的行为都可以获得正确的结果，我们就说这个对象是线程安全**Thread safety**
- If a method or class object can be used by multiple threads at a time without any race condition, then the class is thread-safe. Thread safety is used to make a program safe to use in multithreaded programming.
- 堆是进程和线程共有的空间，分全局堆和局部堆。全局堆就是所有没有分配的空间，局部堆就是用户分配的空间。堆在操作系统对进程初始化的时候分配，运行过程中也可以向系统要额外的堆，但是用完了要还给操作系统，要不然就是内存泄漏。在Java中，堆是Java虚拟机所管理的内存中最大的一块，是所有线程共享的一块内存区域，在虚拟机启动时创建。堆所存在的内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。
- 栈是每个线程独有的，保存其运行状态和局部自动变量的。栈在线程开始的时候初始化，每个线程的栈互相独立，因此，栈是线程安全的。操作系统在切换线程的时候会自动切换栈。栈空间不需要在高级语言里面显式的分配和释放。
- 目前主流操作系统都是多任务的，即多个进程同时运行。为了保证安全，每个进程只能访问分配给自己的内存空间，而不能访问别的进程的，这是由操作系统保障的。在每个进程的内存空间中都会有一块特殊的公共区域，通常称为堆（内存）。进程内的所有线程都可以访问到该区域，这就是造成问题的潜在原因
- 实现线程安全：
    - Synchronization
    - Using Volatile keyword
    - Using a lock based mechanism
    - Use of atomic wrapper classes

013. 守护线程的理解

- 守护线程 daemon thread：为所有非守护线程提供服务的线程；任何一个守护线程都是整个JVM中所有非守护线程的保姆；守护线程的生死无关重要，它却依赖整个进程而运行；如果其他线程结束了，没有要执行的了，程序就结束了，守护线程也就中断了；
- 注意： 由于守护线程的终止是自身无法控制的，因此千万不要把IO、File等重要操作逻辑分配给它；因为它不靠谱；

- 守护线程的作用是什么?
  - 举例， GC垃圾回收线程：就是一个经典的守护线程，当我们的程序中不再有任何运行的Thread,程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是JVM上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源
- 应用场景
  - 来为其它线程提供服务支持的情况
  - 或者在任何情况下，程序结束时，这个线程必须正常且立刻关闭，就可以作为守护线程来使用；反之，如果一个正在执行某个操作的线程必须要正确地关闭掉否则就会出现不好的后果的话，那么这个线程就不能是守护线程，而是用户线程。通常都是些关键的事务，比方说，数据库录入或者更新，这些操作都是不能中断的
  - thread.setDaemon(true)必须在thread.start()之前设置，否则会跑出一个IllegalThreadStateException异常
  - 不能把正在运行的常规线程设置为守护线程
  - 在Daemon线程中产生的新线程也是Daemon的
  - 守护线程不能用于去访问固有资源，比如读写操作或者计算逻辑。因为它会在任何时候甚至在一个操作的中间发生中断
  - Java自带的多线程框架，比如ExecutorService，会将守护线程转换为用户线程，所以如果要使用后台线程就不能用Java的线程池

014. Java死锁如何避免

- **Deadlock** is a situation in which every thread is waiting for a resource which is held by some other waiting thread.
- 造成死锁的几个原因
  - 一个资源每次只能被一个线程使用
  - 一个线程在阻塞等待某个资源时，不释放已占有资源
  - 一个线程已经获得的资源，在未使用完之前，不能被强行剥夺
  - 若干线程形成头尾相接的循环等待资源关系
- 这是造成死锁必须要达到的4个条件，如果要避免死锁，只需要不满足其中某一个条件即可。而其中前3个条件是作为锁要符合的条件，所以要避免死锁就需要打破第4个条件，不出现循环等待锁的关系
- 避免死锁的几个方法
  - 注意加锁顺序，保证每个线程按同样的顺序进行加锁, **Avoid Nested lock**, **Avoid unnecessary locks**
  - 注意加锁时限，可以针对所设置一个超时时间，使用lock.tryLock（timeout）来替代使用内部锁机制
  - 注意死锁检查，这是一种预防机制，确保在第一时间发现死锁并进行解决
  - 使用thread join

015. 线程池的用处以及参数

- **Thread pool** represents a group of worker threads, which are waiting for the task to be allocated
- 为什么用线程池
  - 降低资源消耗；提高线程利用率，降低创建和销毁线程的消耗
  - 提高响应速度；任务来了，直接有线程可用可执行，而不是先创建线程，再执行
  - 提高线程的可管理性；线程是稀缺资源，使用线程池可以统一分配调优监控
- 线程池参数
  - corePoolSize 核心线程数：线程池中的基本线程数量，创建后不会消除，是一种常驻线程
  - maximumPoolSize 最大线程数：当阻塞队列满了之后，逐一启动
  - keepAliveTime 最大线程的存活时间：当阻塞队列的任务执行完后，超出核心线程数之外的线程线程的最大空闲存活时间
  - unit 最大线程的存活时间单位
  - workQueue 阻塞队列，用来存放待执行的任务：当核心线程满后，后面来的任务都进入阻塞队列，直到整个队列被放满但任务还再持续进入则会开始创建新的线程
  - threadFactory 线程工厂：生产线程，使用默认的创建工厂，产生的线程都在同一个组内，拥有相同的优先级，且都不是守护线程；也可以通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名
  - handler 线程池的饱和策略：当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务
    - AbortPolicy: 直接抛出异常，默认策略
    - CallerRunsPolicy: 用调用者所在的线程来执行任务
    - DiscardOldestPolicy: 丢弃阻塞队列中靠最前的任务，并执行当前任务
    - DiscardPolicy: 直接丢弃任务

016. 线程池的底层工作原理

- 线程池内部是通过队列+线程实现的，当我们利用线程池执行任务时：
- 使用execute() 方法提交一个Runnable对象
- 如果此时线程池中的线程数量小于corePoolSize，即使线程池中的线程都处于空闲状态，也要创建新的线程来执行Runnable
- 如果此时线程池中的线程数量大于等于corePoolSize，但是缓冲队列workQueue未满，那么Runnable被放入workQueue
- 如果缓冲队列workQueue满了，但是线程池的线程数小于maximumPoolSize，建新的线程来执行Runnable
- 如果大于maximumPoolSize则通过 handler执行拒绝策略，拒绝此Runnable

017. 线程池五种状态

- RUNNING 会接受新任务，会处理队列中的任务
- SHUTDOWN 不会接受新任务，会处理队列中的任务，处理完中断所有线程
- STOP 不会接受新任务，不会处理队列中的任务，直接中断所有线程
- TIDYING 所有线程都停止后，转换为Tidying ，一旦达到此状态就会调用Terminated()
- TERMINATED： Terminated()执行完转换为TERMINATED

018. 线程池中阻塞队列的作用

- 一般的队列只能保证作为一个有限长度的缓冲区，如果超出了缓冲长度，就无法保留当前的任务了，阻塞队列通过阻塞可以保留住当前想要继续入队的任务。阻塞队列可以保证任务队列中没有任务时阻塞获取任务的线程，使得线程进入wait状态，释放cpu资源。阻塞队列自带阻塞和唤醒的功能，不需要额外处理，无任务执行时,线程池利用阻塞队列的take方法挂起，从而维持核心线程的存活、不至于一直占用cpu资源
- 在创建新线程的时候，是要获取全局锁的，这个时候其它的就得阻塞，影响了整体效率
- **为什么是先添加列队而不是先创建最大线程**
    - 就好比一个企业里面有10个（core）正式工的名额，最多招10个正式工，要是任务超过正式工人数（task > core）的情况下，工厂领导（线程池）不是首先扩招工人，还是这10人，但是任务可以稍微积压一下，即先放到队列去（代价低）。10个正式工慢慢干，迟早会干完的，要是任务还在继续增加，超过正式工的加班忍耐极限了（队列满了），就的招外包帮忙了（注意是临时工）要是正式工加上外包还是不能完成任务，那新来的任务就会被领导拒绝了（线程池的拒绝策略）

019. 线程池中线程复用原理

- 线程池将线程和任务进行解耦，线程是线程，任务是任务，摆脱了之前通过 Thread 创建线程时的一个线程必须对应一个任务的限制
- 在线程池中，同一个线程可以从阻塞队列中不断获取新任务来执行，其核心原理在于线程池对 Thread 进行了封装，并不是每次执行任务都会调用 Thread.start() 来创建新线程，而是让每个线程去执行一个“循环任务”，在这个“循环任务”中不停检查是否有任务需要被执行，如果有则直接执行，也就是调用任务中的 run 方法，将 run 方法当成一个普通的方法执行，通过这种方式只使用固定的线程就将所有任务的 run 方法串联起来

020. ExecutorService 理解

- The Executor Interface provided by the package java.util.concurrent is the simple interface used to execute the new task.
- The execute() method of Executor interface is used to execute some given command.
- The ExecutorService Interface is the subinterface of Executor interface and adds the features to manage the lifecycle.
- `ExecutorService`接口表示线程池

```
// 常用实现类有 FixedThreadPool/ CachedThreadPool/ SingleThreadExecutor
ExecutorService executor = Executors.newFixedThreadPool(3);
executor.submit(new Thread())
executor.shutdown();
```

- 需要定期反复执行任务使用`ScheduledThreadPool`

```
ScheduledExecutorService ses = Executors.newScheduledThreadPool(4);
// 2秒后开始执行定时任务，每3秒执行:
ses.scheduleAtFixedRate(new Task("fixed-rate"), 2, 3, TimeUnit.SECONDS);
// 2秒后开始执行定时任务，以3秒为间隔执行:
ses.scheduleWithFixedDelay(new Task("fixed-delay"), 2, 3, TimeUnit.SECONDS);
```

021. FutureTask 理解

- Java Future interface gives the result of a concurrent process. The Callable interface returns the object of java.util.concurrent.Future.
- Java FutureTask class provides a base implementation of the Future interface.
  - The result can only be obtained if the execution of one task is completed, and if the computation is not achieved then get method will be blocked.
  - If the execution is completed, then it cannot be re-started and can't be canceled.
- `ExecutorService.submit()`方法返回`Future`类型，一个`Future`类型的实例代表一个未来能获取结果的对象

```
ExecutorService executor = Executors.newFixedThreadPool(4);
// 定义任务:
Callable<String> task = new Task();
// 提交任务并获得Future:
Future<String> future = executor.submit(task);
// 从Future获取异步执行返回的结果:
String result = future.get(); // 可能阻塞
```

- 提交一个`Callable`任务后，会同时获得一个`Future`对象，在主线程调用`Future`对象的`get()`方法，可以获得异步执行的结果

022. 什么是原子操作

- The Atomic action is the operation which can be performed in a single unit of a task without any interference of the other operations.
- The Atomic action cannot be stopped in between the task. Once started it will stop after the completion of the task only.
- An increment operation such as `i++` does not allow an atomic action.
- All reads and writes operation for the primitive variable (except long and double) are the atomic operation.

- All reads and writes operation for the volatile variable (including long and double) are the atomic operation.
- The Atomic methods are available in java.util.Concurrent package.

023. lock interface of JUC

- The java.util.concurrent.locks.Lock interface is used as the synchronization mechanism.
- It works similar to the synchronized block, but with a few differences
    - Lock interface provides the **option of timeout** whereas the synchronized block doesn't provide that.
    - The methods of Lock interface, i.e., Lock() and Unlock() can be called in different methods whereas single synchronized block must be fully contained in a single method.

024. ReentrantLock中的公平锁和非公平锁的底层实现

- 首先不管是公平锁和非公平锁，它们的底层实现都会使用AQS来进行排队
- 公平锁和非公平锁的区别在于线程在使用lock()方法加锁时
    - 如果是公平锁，会先检查AQS队列中是否存在线程在排队，如果有线程在排队，则当前线程也进行排队
    - 如果是非公平锁，则不会去检查是否有线程在排队，而是用CAS抢线程，否则使用acquire公平获取锁
- 不管是公平锁还是非公平锁，一旦没竞争到锁，都会进行排队，当锁释放时，都是唤醒排在最前面的线程，所以非公平锁只是体现在了线程加锁阶段，而没有体现在线程被唤醒阶段
- 另外，ReentrantLock是可重入锁，不管是公平锁还是非公平锁都是可重入的

025 CountDownLatch和Semaphore的区别和底层原理

- CountDownLatch表示计数器，可以给CountDownLatch设置一个数字，一个线程调用CountDownLatch的await()将会阻塞，其他线程可以调用CountDownLatch的countDown()方法来对CountDownLatch中的数字减一，当数字被减成0后，所有await的线程都将被唤醒。
- 对应的底层原理就是，调用await()方法的线程会利用AQS排队，一旦数字被减为0，则会将AQS中排队的线程依次唤醒
- Semaphore表示信号量，可以设置许可的个数，表示同时允许最多多少个线程使用该信号量，通过acquire()来获取许可，如果没有许可可用则线程阻塞，并通过AQS来排队，可以通过release()方法来释放许可，当某个线程释放了某个许可后，会从AQS中正在排队的第一个线程开始依次唤醒，直到没有空闲许可

026. 对AQS的理解 AQS如何实现可重入锁

- AQS是一个JAVA线程同步的框架, 是JDK中很多锁工具的核心实现框架
- 在AQS中，维护了一个int类型的信号量state和一个线程组成的双向链表队列。其中线程队列是用来给线程排队的，而state就像是一个红绿灯，用来控制线程排队或者放行的。

- 在可重入锁这个场景下，state就用来表示加锁的次数。0表示无锁，每加一次锁，state就加1。释放锁state就减1

027. ThreadLocal详解

- ThreadLocal原理
  - ThreadLocal是Java中所提供的线程本地存储机制，可以利用该机制将数据缓存在某个线程内部，该线程可以在任意时刻、任意方法中获取缓存的数据
  - 每个Thread都有一个ThreadLocalMap类型的threadLocals，key为ThreadLocal对象，value为要缓存的值
  - ThreadLocalMap由一个个entry组成，entry key是ThreadLocal对象，而且是弱引用
  - ThreadLocal会首先获取当前线程对象t, 然后从线程t中获取到ThreadLocalMap对象threadLocals
  - 如果当前线程的threadLocals已经初始化(即不为null)，但是不存在以当前ThreadLocal对象为Key的的对象, 那么重新创建一个Key，并且添加到当前线程的threadLocals Map中
  - 如果当前线程的threadLocals已经初始化(即不为null)，并且存在以当前ThreadLocal对象为Key的值, 则直接返回当前线程要获取的对象;
  - 如果当前线程的threadLocals属性还没有被初始化, 则重新创建一个ThreadLocalMap对象, 并且创建一个Key并添加到ThreadLocalMap对象中
- ThreadLocal造成内存泄露的问题
  - 内存泄漏: 程序中已动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费
  - 如果用线程池来操作ThreadLocal 对象确实会造成内存泄露, 因为对于线程池里面不会销毁的线程, 里面总会存在着<ThreadLocal, LocalVariable>的强引用
    - 强引用：使用new以及反射创建具有强引用，不会被回收，jvm宁愿OOM也不会回收
    - 弱引用：无论内存是否充足，都会被回收
  - 因为final static 修饰的 ThreadLocal 并不会释放, 而ThreadLocalMap 对于 Key 虽然是弱引用, 但是强引用不会释放, 弱引用当然也会一直有值，同时创建的LocalVariable对象也不会释放, ThreadLocalMap 和Thread生命周期一样长，就造成了内存泄露; 如果LocalVariable对象不是一个大对象的话, 其实泄露的并不严重
  - 泄露的内存 = 核心线程数 * LocalVariable对象的大小
  - 所以, 为了避免出现内存泄露的情况, ThreadLocal提供了一个清除线程中对象的方法, 即 remove, 其实内部实现就是调用 ThreadLocalMap 的remove方法
  - 找到Key对应的Entry, 并且清除Entry的Key(ThreadLocal)置空, 随后清除过期的Entry即可避免内存泄露

9. Java JVM

001. Understanding of Java virtual machine

- **Java Virtual Machine** is a virtual machine that enables the computer to **run the Java program**.
- JVM acts like a run-time engine which calls the main method present in the Java code. JVM is the specification which must be implemented in the computer system. The Java code is compiled by JVM to be a Bytecode which is machine independent and close to the native code.

002. 什么是字节码 采用字节码的好处

- Java中的编译器和解释器
    - Java中引入了虚拟机的概念，即在机器和编译程序之间加入了一层抽象的虚拟的机器。这台虚拟的机器在任何平台上都提供给编译程序一个的共同的接口。编译程序只需要面向虚拟机，生成虚拟机能够理解的代码，然后由解释器来将虚拟机代码转换为特定系统的机器码执行。在Java中，这种供虚拟机理解的代码叫做 字节码 (Bytecode, 扩展名为 .class的文件)，它不面向任何特定的处理器，只面向虚拟机
- 每一种平台的解释器是不同的，但是实现的虚拟机是相同的。Java源程序经过编译器编译后变成字节码，字节码由虚拟机解释执行，虚拟机将每一条要执行的字节码送给解释器，解释器将其翻译成特定机器上的机器码，然后在特定的机器上运行。这也就是解释了Java的编译与解释并存的特点
- Java Code -> class loader, compiler -> Bytecode file -> JVM -> interpreter -> Machine Code -> 程序运行
- 采用字节码的好处：Java语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以Java程序运行时比较高效，而且，由于字节码并不专对一种特定的机器，因此，Java程序无须重新编译便可在多种不同的计算机上运行

003. Types of memory areas are allocated by JVM

- **Heap:** It is the runtime data area in which the memory is allocated to the objects
- **Method Area:** Method Area stores per-class structures such as the runtime constant pool, field, method data, and the code for methods.
- **Stack**: Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as the thread. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.
- **Native Method Stack:** It contains all the native methods used in the application.
- **Program Counter Register:** PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

004. JVM中哪些是线程共享区

- 堆区 Heap、方法区 Mathod Area 是所有线程共享的
- 虚拟机栈 JVM stacks、本地方法栈 Native Mathod Area、程序计数器 Progarm Counter Register 是每个线程独有的

005. What is classloader

- Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.
- **Bootstrap ClassLoader**: This is the first classloader which is the superclass of Extension classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes, etc.
- **Extension ClassLoader**: This is the child classloader of Bootstrap and parent classloader of Application classloader. It loads the jar files located inside *$JAVA_HOME/jre/lib/ext* directory.
- **Application ClassLoader**: This is the child classloader of Extension classloader. It loads the class files from the classpath. By default, the classpath is set to the current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

006. Java中有哪些类加载器

- JVM中存在三个默认类加载器：bootstrapClassLoader、ExtClassLoader、AppClassLoader
- BootStrapClassLoader 是ExtensionClassLoader 的父类加载器，默认负责加载 JAVA_HOME/lib下的jar包和class文件
- ExtensionClassLoader 是AppClassLoader 的父类加载器，负责加载 JAVA_HOME/lib/ext文件夹下的jar包和class类
- AppClassLoader 是自定义类加载器的父类，负责加载应用程序 classpath下的 jar和class文件

007. 说说类加载器双亲委派模型

- 双亲委派指得是，JVM类加载器加载时，会先委派给ExtClass和Bootstrap父类加载器进行加载，只有当父类加载器反馈无法完成这个加载才会自己尝试加载
- JVM在加载一个类时
  - 会调用 `AppClassLoader` 的loadClass方法来加载这个类，不过会先使用 `ExtClassLoader` 的loadClass方法来加载类
  - 同样 `ExtClassLoader` 的loadClass方法中会先使用 `BootstrapClassLoader` 来加载类
  - 如果 `BootstrapClassLoader` 加载到了就直接成功
  - 如果 `BootstrapClassLoader` 没有加载到，那么 `ExtClassLoader` 就会自己尝试加载该类
  - 如果没有加载到，那么则会由 `AppClassLoader` 来加载这个类
  - 如果 `AppClassLoader` 也加载失败，则会报出异常 `ClassNotFoundException`
- 为什么需要双亲委派
  - 哪一个类加载器都要加载Object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。如果由各个类加载器自行去加载，如果用户自己编写一个object类并且放到了路径当中，那系统就会出现多个object类，java类型体系就混乱了
  - 考虑到安全因素，启动类加载器在核心 Java API发现传递的类已被加载，会直接返回已加载过的类，防止核心API库被随意篡改
  - 自定义了类加载器，并且前行用defindCLass() 方法去加载也不会成功，会抛出一个异常

    - 系统类防止内存中出现多份同样的字节码
    - 保证Java程序安全稳定运行
- 为什么需要破坏双亲委派
    - 在某些情况下父类加载器需要委托子类加载器去加载class文件，受到加载范围的限制，父类加载器无法加载到需要的文件
    - 可能需要部署多个应用程序，不同的应用程序可能会依赖同一个第三方类库的不同版本
    - 如何打破：自定义类加载器，继承ClassLoader，重写loadClass方法，从而改变了类加载器的使用顺序。比如我可以让自己的加载器先尝试加载；找不到再委托父类

008. 什么是Garbage collection

- Garbage collection is a process of reclaiming the unused runtime objects, It is performed for memory management.
- It is the process of removing unused objects from the memory to free up space and make this space available for JVM.
- In Java, it is performed automatically, so java provides better memory management.
- The gc() method is used to **invoke the garbage collector** for cleanup processing.
    - This function explicitly makes the Java Virtual Machine free up the space occupied by the unused objects so that it can be utilized or reused.

```
Map<Integer, Integer> s1 = new HashMap<>();
s1 = null;
System.gc();
```

- Garbage collection is managed by JVM.
    - It is performed when there is not enough space in the memory and memory is running low.

009. GC如何判断对象可以被回收

- JVM GC只回收堆区和方法区
- **引用计数法 (Reference Counting)**
    - 每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收
- **可达性分析法 (Reachability Analysis)**
    - 从 GC Roots 开始向下搜索，搜索所走过的路径称为**引用链(Reference Chain)**。当一个对象到 GCRoots 没有任何引用链相连时，则证明此对象是不可用的，那么虚拟机就判断是可回收对象
- GC Roots的对象有：
    - 虚拟机栈(栈帧中的本地变量表）中引用的对象
    - 方法区中类静态属性引用的对象
    - 方法区中常量引用的对象
    - 本地方法栈中JNI(即一般说的Native方法)引用的对象

- 可达性算法中的不可达对象并不是立即死亡的，对象拥有一次自我拯救的机会。对象被系统宣告死亡至少要经历两次标记过程
    - 第一次是经过可达性分析发现没有与GC Roots相连接的引用链
    - 第二次是在由虚拟机自动建立的Finalizer队列中判断是否需要执行finalize()方法，对象将会被放置在一个名为F-Queue的队列之中，并在稍后由一条虚拟机自动建立的、低优先级的Finalizer线程去执行
    - 执行finalize方法完毕后，GC会再次判断该对象是否可达，若不可达，则进行回收，否则，对象“复活”

010. What is finalize()

- The finalize() method is invoked just before the object is garbage collected.
- It is used to **perform cleanup processing**.
- The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created an object without new, you can use the finalize method to perform cleanup processing (destroying remaining objects).
- The cleanup processing is the process to free up all the resources, network which was previously used and no longer needed.
- finalize() is method of object class hence every object can call finalize() as object class is the superclass of every class in java.

011. JVM有哪些垃圾回收算法

- **MarkSweep 标记清除算法**
    - 这个算法分为两个阶段，标记阶段：把垃圾内存标记出来，清除阶段：直接将垃圾内存回收
    - 这种算法是比较简单的，但是有个很严重的问题，就是会**产生大量的内存碎片**
- **Copying 拷贝算法**
    - 为了解决标记清除算法的内存碎片问题，就产生了拷贝算法。拷贝算法将内存分为大小相等的两半，每次只使用其中一半。垃圾回收时，将当前这一块的存活对象全部拷贝到另一半，然后当前这一半内存就可以直接清除
    - 这种算法没有内存碎片，但是他的问题就在于**浪费空间。**而且，他的效率跟存活对象的个数有关
- **MarkCompack 标记压缩算法**
    - 为了解决拷贝算法的缺陷，就提出了标记压缩算法。这种算法在标记阶段跟标记清除算法是一样的，但是在完成标记之后，不是直接清理垃圾内存，而是将存活对象往一端移动，然后将端边界以外的所有内存直接清除
    - 标记压缩算法是在标记清除算法之上，又进行了对象的移动排序整理，因此**成本更高，却解决了内存碎片问题**
- 分代回收

012. 什么是STW

- STW: Stop-The-World，是在垃圾回收算法执行过程当中，需要将JVM内存冻结的一种状态
- 在STW状态下，JAVA的所有线程都是停止执行的-GC线程除外，native方法可以执行，但是，不能与JVM交互
- GC各种算法优化的重点，就是减少STW，同时这也是JVM调优的重点

013. JVM有哪些垃圾回收器

- 新生代收集器
  - Serial
  - ParNew
  - Parallel Scavenge
- 老生代收集器
  - CMS
  - Serial Old
  - Parallel Old
- 整堆收集器：
  - G1

014. 分代回收

- 新生代：绝大多数最新被创建的对象都会被分配到这里，由于大部分在创建后很快变得不可达，很多对象被创建在新生代，然后"消失"。对象从这个区域"消失"的过程称之为 **Minor GC**
- 老生代：对象没有变得不可达，并且从新生代周期中存活了下来，会被拷贝到这里。其区域分配的空间要比新生代多。也正由于其相对大的空间，发生在老生代的GC次数要比新生代少得多。对象从老生代中消失的过程，称之为 **Major GC 或者 Full GC**
- 新生代的构成
  - 用来保存那些第一次被创建的对象，它被分成三个空间：一个伊甸园空间& 两个幸存者空间(From Survivor、To Survivor)
  - 默认新生代空间的分配：Eden : From : To = 8 : 1 : 1
  - 绝大多数刚刚被创建的对象会存放在伊甸园空间
  - 在伊甸园执行第一次Minor GC之后，存活的对象被移动到其中一个幸存者空间
  - 再次Minor GC时，将Eden区和Survivor From 区中存活的对象复制到Survivor To区之中 (如果对象的年龄到达了老生代的标准时则赋值到老生代(通常15)) 然后清空Eden区和Survivor From 区，最后将 Survivor To区和Survivor From 区互换
  - 很多次之后，Suvivor To区满了，就将所有对象移入老生代
- 触发Minor GC：当Eden区满时
- 触发Full GC：老生代会在新生代对象转入及创建大对象、大数组时会触发空间不足时, 或者System.gc()方法的调用
- 新生代都是复制算法

    - 新生代对象生存时间比较短，80%都是要回收的对象，采用标记清除算法则内存空间碎片化严重，采用复制算法灵活高效，且便于整理空间
- 老生代都是标记整理算法
    - 标记整理算法解决来标记清除算法的内存碎片化的问题，又解决了复制算法的两个Survivor区的问题，因为老生代的空间比较大，不可能采用复制算法，特别占用内存空间

015. CMS GC

- Concurrent Mark Sweep收集器是一种年老代垃圾收集器，其最主要目标是获取最短垃圾回收停顿时间，使用多线程的标记清除算法
- 整个过程分为以下 4 个阶段
    - 初始标记：只是标记一下 GC Roots 能直接关联的对象，速度很快，需要暂停所有的工作线程
    - 并发标记：进行 GC Roots 跟踪的过程，和用户线程一起工作，不需要暂停工作线程
    - 预清理阶段（针对老生代内的引用变化）是和用户线程并发的，为的是减轻下一阶段（会STW）的工作量。用来处理前一个阶段因为引用关系改变导致没有标记到的存活对象的，它会扫描所有标记为Dirty的Card
    - 重新标记（针对年轻代对老生代的引用变化）（会STW）：这个阶段会导致第二次stop the word，该阶段的任务是完成标记整个年老代的所有的存活对象，由于之前的预处理阶段是与用户线程并发执行的，这时候可能年轻代的对象对老生代的引用已经发生了改变
- 并发清除
    - 清除 GC Roots 不可达对象，和用户线程一起工作，不需要暂停工作线程。由于耗时最长的并发标记和并发清除过程中，gc线程可以和用户线程并发工作
- 并发清除是如何保证安全的
    - 并发清理阶段，黑色对象不可能直接（不过经过灰色对象）指向某个白色对象，新的对象也是无法指向某个白色对象的
- CMS的缺点
    - 浮动垃圾：由于CMS并发清理阶段用户线程还在运行着，伴随程序运行会有新垃圾产生，所以CMS只能下一次GC清理掉
    - 有空间碎片：CMS基于标记-清除,所以会产生空间碎片。CMS提供了-XX:UseCMSCompactAtFullCollection开发参数用于开启内存碎片的合并整理，由于内存整理是无法并行的，所以停顿时间会变长

016. G1 GC

- STW更可控，G1在停顿时间上添加了预测机制，优先回收“收益大”的region，用户可以指定期望停顿时间。
- G1 将整个堆划分为一个个大小相等的region（1~32MB），每一块region的内存是连续的。在每个分区内部又被分成了若干个大小为512 Byte卡片(Card)
- Remember Set
    - 每个Region被分成了多个Card,Rset记录了其他Region中的对象引用本Region中对象
- RSet怎么辅助GC的呢?

    - 在做YGC的时候，只需要选定young generation region的RSet作为根集，这些RSet记录了old->young的跨代引用，避免了扫描整个old generation。 而mixed gc的时候，old generation中记录了old->old的RSet，young->old的引用由扫描全部young generation region得到，这样也不用扫描全部old generation region。所以RSet的引入大大减少了GC的工作量。
- G1中提供了两种垃圾回收模式，Young GC和Mixed GC，两种都是STW
    - Young GC：**选定所有年轻代里的Region**。通过控制年轻代的region个数，来控制young GC的时间开销。每次young gc会回收所有Eden以及Survivor区，并且将存活对象复制到Old区以及另一部分的Survivor区。
    - Mixed GC：选**定所有年轻代里的Region**，外加根据global concurrent marking统计得出收集收益高的**若干老生代Region**。在用户指定的开销目标范围内尽可能选择收益高的老生代Region。
    - Mixed GC不是full GC，它只能回收部分老生代的Region，如果mixed GC实在无法跟上程序分配内存的速度，导致老生代填满无法继续进行Mixed GC，就会使用serial old GC（full GC）来收集整个GC heap。G1是不提供full GC的。
- G1的回收过程
  1）初始标记
  是STW的，标记出从GC Root开始直接可达的对象。
  2）并发标记
  从GC Roots开始对堆中对象进行可达性分析，找出存活对象，耗时较长。
  3）最终标记
  标记那些在并发标记阶段发生变化的对象，将被回收。
  4）筛选回收
  首先对各个Regin的回收价值和成本进行排序，根据用户所期待的GC停顿时间指定回收计划，回收一部分Region。把Region中存活的对象复制到其他Region中，把这些已经被回收的Region直接放到空闲列表中

017. 垃圾回收分为哪些阶段

- GC分为四个阶段
    - 第一：初始标记 标记出GCRoot直接引用的对象。STW
    - 第二：标记Region，通过RSet标记出上一个阶段标记的Region引用到的Old区Region
    - 第三：并发标记阶段：跟CMS的步骤是差不多的。只是遍历的范围不再是整个Old区，而只需要遍历第二步标记出来的Region
    - 第四：重新标记： 跟CMS中的重新标记过程是差不多的。
    - 第五：垃圾清理：与CMS不同的是，G1可以采用拷贝算法，直接将整个Region中的对象拷贝到另一个Region。而这个阶段，G1只选择垃圾较多的Region来清理，并不是完全清理

018. 什么是三色标记

- 三色标记法是一种垃圾回收法，它可以让JVM不发生或仅短时间发生STW(Stop The World)，从而达到清除JVM内存垃圾的目的
- JVM中的CMS、G1垃圾回收器所使用垃圾回收算法即为三色标记法
- 三色标记法将内存对象的颜色分为了黑、灰、白，三种颜色
    - 黑色：自己已经被标记过，且该对象下的成员变量也全部都被标记过

    - 灰色：自己已经被标记过，但该对象下的成员变量没有全被标记完
    - 白色：自己没有被标记过

019. 项目如何排查JVM问题

- 对于还在正常运行的系统
    - 可以使用jmap来查看JVM中各个区域的使用情况
    - 可以通过jstack来查看线程的运行情况，比如哪些线程阻塞、是否出现了死锁
    - 可以通过jstat命令来查看垃圾回收的情况，特别是fullgc，如果发现fullgc比较频繁，那么就得进行调优了
    - 通过各个命令的结果，或者jvisualvm等工具来进行分析
    - 首先，初步猜测频繁发送fullgc的原因，如果频繁发生fullgc但是又一直没有出现内存溢出，那么表示fullgc实际上是回收了很多对象了，所以这些对象最好能在younggc过程中就直接回收掉，避免这些对象进入到老生代，对于这种情况，就要考虑这些存活时间不长的对象是不是比较大，导致年 轻代放不下，直接进入到了老生代，尝试加大年轻代的大小，如果改完之后，fullgc减少，则证明修改有效
    - 同时，还可以找到占用CPU最多的线程，定位到具体的方法，优化这个方法的执行，看是否能避免某些对象的创建，从而节省内存
- 对于已经发生了OOM的系统
    - 一般生产系统中都会设置当系统发生了OOM时，生成当时的dump文件（-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/usr/local/base)
    - 我们可以利用jsisualvm等工具来分析dump文件
    - 根据dump文件找到异常的实例对象，和异常的线程（占用CPU高），定位到具体的代码
    - 然后再进行详细的分析和调试
- 需要分析、推理、实践、总结、再分析，最终定位到具体的问题

10. Java 8 New Features

10.1 Lambda Expression

- 使用匿名内部类问题
    - 需要实现接口并实现抽象方法
    - 为了指定的方法体，不得不需要接口的实现类
    - 为了省去定义一个实现类的麻烦，不得不使用匿名内部类
    - 需要覆盖重写抽象方法，然而只有方法体才是关键
- 什么是Lambda表达式
    - Lambda 表达式是一个匿名函数, 直接对应于其中的 Lambda 抽象
    - Lambda 表达式理解为是一段可以传递的代码, 也可以理解为函数式编程, 将一个函数作为参数进行传递

- Lambda 表达式的基本语法

```
// (参数类型 参数名称) -> { 代码体; }
(parameters) -> expression
(parameters) -> {statements;}
```

- 优点
  - 只需要将要执行的代码放到一个Lambda表达式中，不需要定义类，不需要创建对象
  - 简化匿名内部类的使用，语法更加简单
- 代码示例
  - 传统写法

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("新线程任务执行");
    }
}).start();
```

  - lambda写法

```
new Thread(
    () -> System.out.println("新线程任务执行")
).start();
```

- Lambda标准格式省略写法
  - 参数名字不用跟定义一样，因为参数类型推断 Type inference，箭头左侧参数不用写类型
  - 如果小括号内有且仅有一个参数，则小括号可以省略
  - 如果大括号内有且仅有一个语句，可以同时省略大括号、return关键字及语句分号
- Lambda表达式使用条件
  - 方法的参数或局部变量类型必须为接口才能使用Lambda
  - 使用函数式接口：接口中有且仅有一个抽象方法, 注解 @FunctionalInterface
- 内置函数式接口
  - Supplier接口
    - 生产一个数据，无参有get方法返回的接口

```
@FunctionalInterface
public interface Supplier<T> {
    public abstract T get();
}
```

  - Consumer接口
    - 消费一个数据，有参无返回的接口

```java
@FunctionalInterface
public interface Consumer<T> {
    public abstract void accept(T t);
}
```

- Function接口
  - 根据一个类型的数据得到另一个类型的数据，有参数有返回值

```java
@FunctionalInterface
public interface Function<T, R> {
    public abstract R apply(T t);
}
```

- Predicate接口
  - 对某种类型的数据进行判断，得到一个boolean值结果

```java
@FunctionalInterface
public interface Predicate<T> {
    public abstract boolean test(T t);
}
```

10.2 Default and Static Methods in Interfaces

- JDK 8以前的接口
  - 接口只能有**静态常量**和**公共抽象方法**
  - 如果不强制所有实现类创建新方法的实现，就不可能向现有接口添加新功能
- JDK 8增强的接口
  - 接口还可以有**默认方法**和**静态方法**
  - 关键字default可以给接口方法添加默认实现，所有实现类不需要重写方法就可以直接使用
  - 静态方法不能被继承，实现类不能重写接口静态方法，只能使用接口名调用

10.3 Method References

- 方法引用是以类的名字开头，后跟双冒号，后跟方法的名称，其中双冒号 `::` 为方法引用运算符
- 如果Lambda所要实现的方案，已经有其他方法存在相同方案，那么则可以使用**方法引用**
- 方法引用共有4种形式
  - 引用构造方法：ClassName::new
  - 引用类的静态方法：ClassName::staticmethodName
  - 引用类的普通方法：ClassName::methodName
  - 引用对象的方法：instanceName::methodName

10.4 Stream API

- 对集合中的元素进行操作的时候，除了必需的添加、删除、获取外，最典型的就是集合遍历，Java8 新增了Stream类，从而把函数式编程的风格引入到Java语言中
- Stream流类似于工厂车间的“生产流水线”，Stream流不是一种数据结构，不保存数据，而是对数据进行加工处理
- Stream 分类
  - Intermediate Operations: filter, peek, limit, skip, map, flatMap, concat, sorted, distinct
  - Terminal Operations: 不再支持链式调用: forEach, min, max, count, allMatch, anyMatch, collect, noneMatch
- Create Stream
  - Create stream based on array

    ```
    Integer[] array = {1,2,3,4,5};
    Stream<Integer> arrayStream = Arrays.stream(array);
    ```

  - Create stream based on collection

    ```
    List<String> list = new ArrayList<>();
    Stream<String> listSream = list.stream();
    ```

  - Stream 接口的静态方法 of 可以获取数组对应的流

    ```
    Stream<String> stream = Stream.of("aa", "bb", "cc");
    ```

  - Create stream using generate mathod

    ```
    Stream<Integer> generateStream = Stream.generate(() -> 5);
    ```

  - Create stream using iterate mathod

    ```
    Stream<Integer> iterateStream = Stream.iterate(1, x -> x+2);
    ```

- Stream常用方法
  - map:

    ```
    Stream<String> original = Stream.of("11", "22", "33");
    Stream<Integer> result = original.map(Integer::parseInt);
    ```

- 收集Stream流的结果
  - 对流操作完成之后，如果需要将流的结果保存到数组或集合中，可以收集流中的数据
  - Stream流中的结果到集合中

```
Stream<String> stream = Stream.of("aa", "bb", "cc");

List<String> list = stream.collect(Collectors.toList());
Set<String> set = stream.collect(Collectors.toSet());

ArrayList<String> arrayList =
stream.collect(Collectors.toCollection(ArrayList::new));
HashSet<String> hashSet =
stream.collect(Collectors.toCollection(HashSet::new));
```

  - Stream流中的结果到数组中

```
Stream<String> stream = Stream.of("aa", "bb", "cc");
String[] strings = stream.toArray(String[]::new);
```

- 并行的Stream流
  - 串行的Stream流是在一个线程上执行
  - parallelStream其实就是一个并行执行的流。它通过默认的ForkJoinPool，可能提高多线程任务的速度

```
ArrayList<Integer> list = new ArrayList<>();
// 直接获取并行的流
Stream<Integer> stream = list.parallelStream();
 // 将串行流转成并行流
Stream<Integer> stream = list.stream().parallel()
```

10.5 Optional Class

- Before Java 8
  - we had to validate values referred because of the possibility of throwing the **NullPointerException (NPE)**.
  - java 经常需要使用大量的代码来处理空指针异常，而这种操作往往会降低程序的可读性

```
User recipient = getUserNull("Amy");
if (recipient != null)
    ...
```

- Optional in Java 8
  - Elegant solution for returning null

```
Optional<User> recipient = getUserOptional("Amy");
Optional<User> recipient = Optional.of("Amy");
if (recipient.isPresent())
    recipient.get();
```

- Optional 类是一个可以为null的容器对象, 主要作用为了解决避免Null检查，防止NullPointerException

- 如果此值不为空，它可以返回此对象的值
- 当此容器内的值为 null 时，它不是抛出 NPE而是执行一些预定义的操作，例如`orElse`，`orElseThrow`，`orElseGet`

10.6 Date and Time API

- Dates before JDK1.8
  - Calendar：实现日期和时间字段之间的转换，它的属性是可变的，因此，它不是线程安全的。
  - DateFormat：格式化和分析日期字符串
  - Date：用来承载日期和时间信息，它的属性是可变的，因此，它不是线程安全的。
  - 旧版日期时间设计很差, 非线程安全，时区处理麻烦, JDK 8中增加了一套全新的日期时间API
- Dates in JDK1.8
  - 主要包含了处理日期、时间、日期/时间、时区、时刻 (instants)、和时钟（clock） 等操作
  - 代替 java.util.Date和Calendar，位于 java.time 包
  - Thread safe
  - Time-zone logic included
- LocalDate，LocalTime，LocalDateTime
  - `LocalDate` 表示日期，包含年月日，格式为 2019-10-16

    ```
    LocalDate ld1 = LocalDate.now();
    LocalDate ld2 = LocalDate.of(1985, 9, 23);
    LocalDate ld2 = LocalDate.of(1985, Month.DECEMBER, 23);
    ```

  - `LocalTime`表示时间，包含时分秒，格式为 16:38:54.158549300

    ```
    LocalTime lt1 = LocalTime.now();
    LocalTime lt2 = LocalTime.of(10, 5);
    ```

  - `LocalDateTime` 表示日期时间，包含年月日，时分秒，格式为 2018-09-06T15:33:56.750

    ```
    LocalDateTime ldt1 = LocalDateTime.now();
    LocalDateTime ldt2 = LocalDateTime.of(1985, 9, 23, 10, 5);
    LocalDateTime ldt3 = LocalDateTime.of(ld1, lt1);
    ```

- ZonedDateTime, ZonedId
  - `ZonedDateTime`包含时区的时间

    ```
    ZoneId zid = ZoneId.of("Asia/Shanghai");
    LocalDateTime ldt = LocalDateTime.now();
    ZonedDateTime adt = ZonedDateTime.of(ldt, zid);
    ```

- `Clock` 类通过指定一个时区，可以获取到当前的时刻，日期与时间 `Clock.systemUTC()`

```
Clock clock1 = Clock.systemDefaultZone();
Clock clock2 = Clock.system(ZoneId.of("Asia/Shanghai"));
```

- `DateTimeFormatter` 日期时间格式化类

```
LocalDateTime ldt = LocalDateTime.now();
// Y D d M h H m s z Z
DateTimeFormatter formatter1 = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
DateTimeFormatter formatter2 = DateTimeFormatter.ISO_LOCAL_DATE_TIME;
String format = now.format(formatter2);
```

- `Instant` 时间戳，表示一个特定的时间瞬间

```
Instant now = Instant.now();
//精确到秒
System.out.println(now.getEpochSecond());
//精确到毫秒
System.out.println(now.toEpochMilli());
```

- Comparing dates and times

```
LocalDateTime ldt1 = LocalDateTime.now();
LocalDateTime ldt2 = LocalDateTime.of(2023, 9, 23, 10, 5);
ldt1.compareTo(ldt2); ldt1.isAfter(ldt2); ldt1.isBefore(ldt2);
ldt1.isEqual(ldt2);
```

- `Duration` 用于计算2个时间(LocalTime，时分秒)的距离

```
Duration d = Duration.between(lt1, lt2);
```

- `Period` 用于计算2个日期(LocalDate，年月日)的距离

```
Period p = Period.between(ldt1, ldt2);
```

10.7 Base64

- JDK1.8把Base64 编码添加到了标准类库中

```
String str = "Hello World!";
// 编码
String encodeStr = Base64.getEncoder().encodeToString(str.getBytes());
System.out.println(encodeStr);
//解码
String decodeStr = new String(Base64.getDecoder().decode(encodeStr));
System.out.println(decodeStr);
```

10.8 Annotation

- JDK 8引入了重复注解与类型注解
- 重复注解允许在同一个地方多次使用同一个注解
- 类型注解`TYPE_PARAMETER` `TYPE_USE`
  - TYPE_PARAMETER： 表示该注解能写在类型参数的声明语句中
  - TYPE_USE： 表示注解可以再任何用到类型的地方使用。

10.9 参数名字

- 编译的时候增加 -parameters 选项，以及增加反射API与 Parameter,getName() 方法实现了获取方法参数名的功能

```
System.out.println("parameter::" + parameter.getName());
```

- 如果使用指令 javac Test6.java 来编译以上程序，那么运行的结果是 parameter::args()

  如果使用的是 javac Test6.java -parameters 来编译 那么结果是parameter::args

10.10 调用JavaScript

- JDK1.8增加API使得通过Java程序来调用JavaScript代码

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("JavaScript");
System.out.println(engine.getClass().getName());
System.out.println(engine.eval("function f() {return 'hello';}; f() + '
world!';"));
```

10.11 并行数组

- JDK1.8增加了对数组并行处理的方法（parallelXxx）

```
long startParallel = System.currentTimeMillis();
Arrays.parallelSort(simpleArr1);
long endParallel = System.currentTimeMillis();
```