

基础面试题Anki

关于赋值和自增自减

```
package com.atguigu.test;

public class Test {

    public static void main(String[] args) {
        int i = 1;
        i = i++;
        int j = i++;
        int k = i + ++i * i++;
        System.out.println("i=" + i);
        System.out.println("j=" + j);
        System.out.println("k=" + k);
    }
}
```

小结

- 赋值=，最后计算
 - =右边的从左到右加载值依次压入操作数栈
 - 实际先算哪个，看运算符优先级
 - 自增、自减操作都是直接修改变量的值，不经过操作数栈
 - 最后的赋值之前，临时结果也是存储在操作数栈中
-
- 建议：《JVM虚拟机规范》关于指令的部分

答案：
<terminated> Test [Java Application]
i=4
j=1
k=11

请创建双重校验单例模式

```
public class Singleton {
    private static Singleton singleton; ② 静态变量保存实例

    private Singleton() { ① 构造器私有化
    }

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

要点

单例的三个要素

- 一是某个类只能有一个实例;
 - ◆ 构造器私有化
- 二是它必须自行创建这个实例;
 - ◆ 含有一个该类的静态变量来保存这个唯一的实例
- 三是它必须自行向整个系统提供这个实例;
 - ◆ 对外提供获取该实例对象的方式:
 - (1) 直接暴露
 - (2) 用静态变量的get方法获取

```
import java.io.IOException;
import java.util.Properties;

public class Singleton3 {
    public static final Singleton3 INSTANCE;
    private String info;

    static{          饿汉式, 不管需不需要直接new
        try {
            Properties pro = new Properties();
            pro.load(Singleton3.class.getClassLoader().getResourceAsStream("single.
                INSTANCE = new Singleton3(pro.getProperty("info"));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    private Singleton3(String info){
        this.info = info;
    }
}
```

饿汉式
第三种

子类初始化的执行顺序



```

1 package com.atguigu.test;
2
3 public class Father{
4     private int i = test();
5     private static int j = method();
6
7     static{
8         System.out.print("(1)");
9     }
10    Father(){
11        System.out.print("(2)");
12    }
13    {
14        System.out.print("(3)");
15    }
16    public int test(){
17        System.out.print("(4)");
18        return 1;
19    }
20    public static int method(){
21        System.out.print("(5)");
22        return 1;
23    }
24 }

```

```

1 package com.atguigu.test;
2
3 public class Son extends Father{
4     private int i = test();
5     private static int j = method();
6
7     static{
8         System.out.print("(6)");
9     }
10    Son(){
11        System.out.print("(7)");
12    }
13    {
14        System.out.print("(8)");
15    }
16    public int test(){
17        System.out.print("(9)");
18        return 1;
19    }
20    public static int method(){
21        System.out.print("(10)");
22        return 1;
23    }
24
25    public static void main(String[] args) {
26        Son s1 = new Son();
27        System.out.println();
28        Son s2 = new Son();
29    }
30 }

```

以
运

类初始化过程

拿小本记下来

- ① 一个类要创建实例需要先加载并初始化该类
 - ◆ main方法所在的类需要先加载和初始化
- ② 一个子类要初始化需要先初始化父类
- ③ 一个类初始化就是执行<clinit>()方法
 - ◆<clinit>()方法由静态类变量显示赋值代码和静态代码块组成
 - ◆类变量显示赋值代码和静态代码块代码从上到下顺序执行
 - ◆<clinit>()方法只执行一次

答案: (5)(1)(10)(6)(9)(3)(2)(9)(8)(7)



实例初始化过程

拿小本记下来



- ① 实例初始化就是执行<init>()方法
 - ◆<init>()方法可能重载有多个，有几个构造器就有几个<init>方法
- ② <init>()方法由非静态实例变量显示赋值代码和非静态代码块、对应构造器代码组成
- ◆非静态实例变量显示赋值代码和非静态代码块代码从上到下顺序执行，而对应构造器的代码最后执行
- ◆每次创建实例对象，调用对应构造器，执行的就是对应的<init>方法
- ◆<init>方法的首行是super()或super(实参列表)，即对应父类的<init>方法

方法的重写Override

拿小本记下来



- ① 哪些方法不可以被重写
 - ◆ final 方法
 - ◆ 静态方法
 - ◆ private 等子类中不可见方法

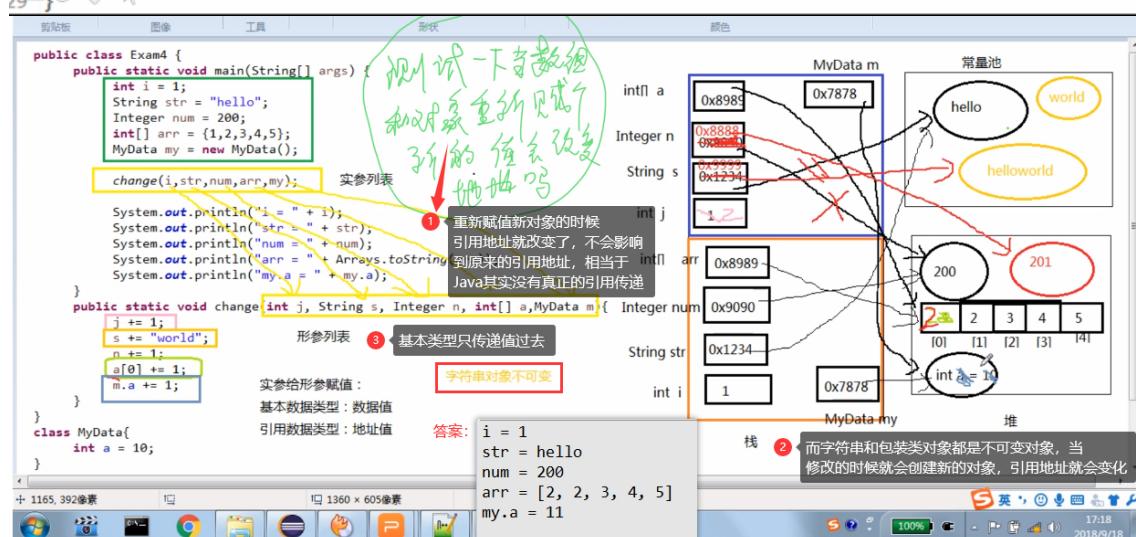
② 对象的多态性

- ◆ 子类如果重写了父类的方法，通过子类对象调用的一定是子类重写过的代码
- ◆ 非静态方法默认的调用对象是 this
- ◆ this 对象在构造器或者说<init>方法中就是正在创建的对象

实参和形参的值传递

```
1 import java.util.Arrays;
2
3 public class Exam4 {
4     public static void main(String[] args) {
5         int i = 1;
6         String str = "hello";
7         Integer num = 2;
8         int[] arr = {1,2,3,4,5};
9         MyData my = new MyData();
10
11         change(i,str,num,arr,my);
12
13         System.out.println("i = " + i);
14         System.out.println("str = " + str);
15         System.out.println("num = " + num);
16         System.out.println("arr = " + Arrays.toString(arr));
17         System.out.println("my.a = " + my.a);
18     }
19     public static void change(int j, String s, Integer n, int[] a, MyData m){
20         j += 1;
21         s += "world";
22         n += 1;
23         a[0] += 1;
24         m.a += 1;
25     }
26 }
27 class MyData{
28     int a = 10;
29 }
```

运行结果：



方法的参数传递机制：

1

① 形参是基本数据类型

- ◆ 传递数据值

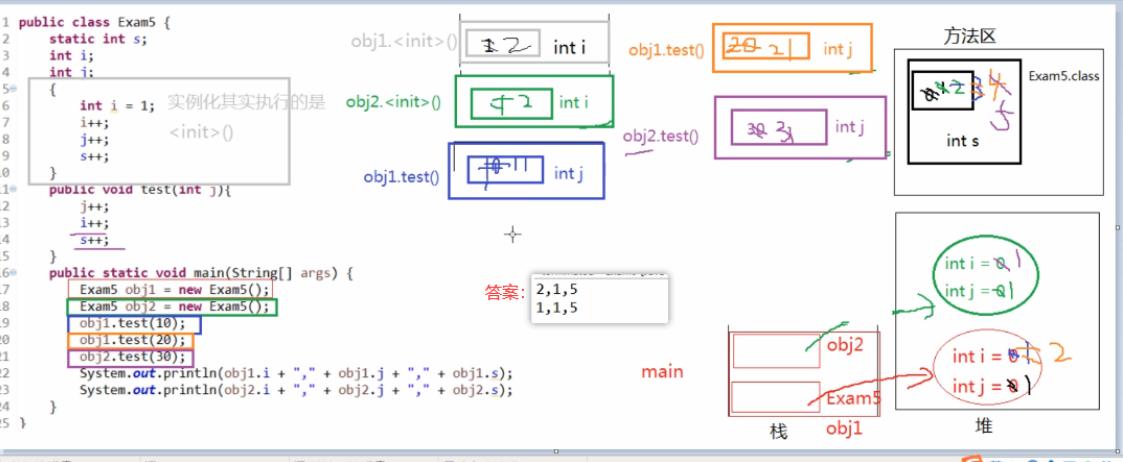
② 实参是引用数据类型

- ◆ 传递地址值
- ◆ 特殊的类型：String、包装类等对象不可变性

成员变量，局部变量，静态变量的区别

```
public class Exam5 {  
    static int s;  
    int i;  
    int j;  
    {  
        int i = 1;  
        i++;  
        j++;  
        s++;  
    }  
    public void test(int j){  
        j++;  
        i++;  
        s++;  
    }  
    public static void main(String[] args) {  
        Exam5 obj1 = new Exam5();  
        Exam5 obj2 = new Exam5();  
        obj1.test(10);  
        obj1.test(20);  
        obj2.test(30);  
        System.out.println(obj1.i + "," + obj1.j + "," + obj1.s);  
        System.out.println(obj2.i + "," + obj2.j + "," + obj2.s);  
    }  
}
```

运行结果：



Spring事务的传播属性

```
//1.请简单介绍Spring支持的常用数据库事务传播属性和事务隔离级别?  
  
/**  
 * 事务的属性:  
 * 1.★propagation: 用来设置事务的传播行为  
 *   事务的传播行为: 一个方法运行在了一个开启了事务的方法中时, 当前方法是使用原来的事务还是开启一个新的事务  
 *   -Propagation.REQUIRED: 默认值, 使用原来的事务  
 *   -Propagation.REQUIRES_NEW: 将原来的事务挂起, 开启一个新的事务  
 * 2.★isolation: 用来设置事务的隔离级别  
 *   -Isolation.REPEATABLE_READ: 可重复读, MySQL默认的隔离级别  
 *   -Isolation.READ_COMMITTED: 读已提交, Oracle默认的隔离级别, 开发时通常使用的隔离级别  
 */  
@Transactional(propagation=Propagation.REQUIRES_NEW)
```

Null, Empty, Blank的区别

- `Null` 指对象没有初始化, 也就是说没有引用指向这个对象
- `isEmpty()`: 此方法可以使用于字符串, 数组, 集合
 - `public boolean isEmpty() {
 return value.length == 0;
}`
 - 根据对象的长度进行判断, 使用这个方法时, 先要排除对象不为null, 当对象为null时, 调用`isEmpty`方法就会报空指针异常。
 - 据源码可知要想要此方法返回true, 也就是一个当对象的长度为0, 如字符串, 数组等, 就会返回true。
 - 需要注意的是当字符串对象内容为空格的时候, 字符串的长度是不为0的, 所以当判断一些字符串需要有数据的时候空格应该也要被否认, 下面有一种方法 (`isNotBlank`) 就是用来处理这种情形的。
- `isNotBlank`: 判断字符串内容不为空串且不为空白串

```
1  * StringUtils.isNotBlank(null) = false  
2  * StringUtils.isNotBlank("") = false  
◦ 3  * StringUtils.isNotBlank(" ") = false 即使有多个空格, 也属于空白串  
4  * StringUtils.isNotBlank("bob") = true  
5  * StringUtils.isNotBlank(" bob ") = true
```

int和Integer之间的比较

```
public static void main(String[] args) {  
    int i1 = 127;  
    Integer i2 = new Integer(127);  
    Integer i3 = 127;  
    Integer i4 = 127;  
    Integer i5 = new Integer(500);  
    Integer i6 = new Integer(500);  
    Integer i7 = 500;  
    Integer i8 = 500;  
    System.out.println(i1 == i2); //true 当与i1基本类型比较的时候i2自动拆箱为基本类型, 相当于基本类型之间的比较  
    System.out.println(i1 == i3); //true 同上
```

```

        System.out.println(i2 == i3); //false 两个不同对象之间的比较
        System.out.println(i3 == i4); //true i3和i4范围在[-128~128)之间的时候且以直接
赋值方式生成对象，那么指向同一个已经缓存好的Integer对象
        System.out.println(i5 == i6); //false 两个不同对象之间的比较
        System.out.println(i7 == i8); //false 范围不在[-128~128)之间，两个不同对象直接
的比
    }
}

```

**问题：为什么Integer当它们范围在[-128, 128) 内的时候且是以 Integer i = xx(xx为一个数
字)直接赋值的方式生成时，返回的对象是同一个对象呢？**

解答：

- 在源码中可以看出一开始就一次性生成了-128到127直接的Integer类型变量存储在cache[]中，**对于-128到127
之间的int类型，返回的都是同一个Integer类型对象。**

- 所以整个工作过程就是：**Integer.class在装载（Java虚拟机启动）时，其内部类型IntegerCache的static块即
开始执行，实例化并暂存数值在-128到127之间的Integer类型对象。当自动装箱int型值在-128到127之间时，
即直接返回IntegerCache中暂存的Integer类型对象。**

- 为什么Java这么设计？我想是出于效率考虑，因为自动装箱经常遇到，尤其是小数值的自动装箱；而如果每次自动装箱都触发new，在堆中分配内存，就显得太慢了；所以不如预先将那些常用的值提前生成好，自动装箱时直接拿出来返回。哪些值是常用的？就是-128到127了。

四大代码块的区别

局部代码块格式如下：

```

1  public class CodeBlock {
2      public static void main(String[] args) {
3          //局部代码块
4          {
5              System.out.println("我是局部代码块");
6              int i = 10;
7              System.out.println(i);
8          }
9          //System.out.println(i); 报错，不存在变量i，说明变量i已经被释放
10     }
11 }
12

```

- 局部代码块：在方法中的代码块，无修饰符
- 作用：控制变量的生命周期（提前释放变量i）提高内存利用率

构造代码块格式如下:

```
1 class Parent {  
2     {  
3         System.out.println("我是构造代码块");  
4     }  
5  
6     public Parent() {  
7         System.out.println("我是构造方法");  
8     }  
9 }  
10
```

- 构造代码块: 在类中且不在方法中的代码块, 无修饰符
- 性质: 创建对象的时候每次调用构造方法前都会先执行构造代码块中的内容
- 作用: 在于可以把多个构造方法中的相同代码抽取出来放在构造代码块中, 对对象进行初始化。

静态代码块格式如下:

```
1 class Parent {  
2     static {  
3         System.out.println("我是静态代码块");  
4     }  
5  
6     public Parent() {  
7         System.out.println("我是构造方法");  
8     }  
9 }  
10
```

- 静态代码块: 在类中且不在方法中的代码块, 修饰符为static
- 性质: 静态代码块最早执行, 类被载入内存时执行, 只执行一次。
- 作用: 静态块常用来执行类属性的初始化
- 静态代码块执行效果如下:

四. 同步代码块

格式如下:

```
1  
2 synchronized(obj)  
3 {  
4     //需要被同步的代码块  
5 }
```

- 同步代码块: obj 称为同步监视器, 也就是锁
- 原理: 当线程开始执行同步代码块前, 必须先获得对同步代码块的锁定。并且任何时刻只能有一个线程可以获得对同步监视器的锁定, 当同步代码块执行完成后, 该线程会释放对该同步监视器的锁定。
- 锁: 其中的锁, 在非静态方法中可为this, 在静态方法中为当前类本身

前后端传值的三个注解

一. @PathVariable 和 @RequestParam注解

- 注解作用：
@PathVariable用于获取路径参数， @RequestParam用于获取查询参数。

例子代码：

```
1 @GetMapping("/user/{Id}")
2 public User getUser(
3     @PathVariable("Id") Long Id,
4     @RequestParam(value = "username", required = false) String username) {
5
6     ...
7 }
8 }
```

- 如上代码：请求URL可以为：/user/123?username=zhangsan那么123会绑定到参数Id，zhangsan绑定到参数username上

二. @RequestBody注解

- 注解作用：
用于读取 Request 请求的 body 部分（且Content-Type 为 application/json 格式）的数据，接收到数据之后会自动将数据绑定到 Java 对象上去。系统会将请求的 body 中的 json 字符串转换为 java 对象。

• 2.1 例子代码：

```
1 @PostMapping("/sign")
2 public ResponseEntity sign(@RequestBody User user) {
3     userService.save(user);
4     return ResponseEntity.ok().build();
5 }
```

- 一个请求方法只可以有一个@RequestBody，但是可以有多个@RequestParam和@PathVariable。

@Bean和@Component的联系和区别

- 最终效果都是为spring容器注册Bean.
- 注册Bean之后都可以使用@Autowired或者@Resource注解注入
- @Component是类级别的注释，而@Bean是方法级别的注释.
- @Bean通常和@Configuration注解搭配使用
- @Bean是方法级别的注释，该方法的名称用作Bean名称
- @Bean显式声明单个Bean，而不是让Spring自动执行。
- 如果想将第三方的类变成组件，又没有源代码，也就没办法使用@Component进行自动配置，这种时候使用@Bean就比较合适了。

- @Component注解一般用于自定义类上将其注入spring容器中进行管理。

Java序列化和序列化UID的作用

为了保存在内存中的各种对象的状态（也就是实例变量，不是方法），并且可以把保存的对象状态再读出来，这是java中的提供的保存对象状态的机制—序列化。将对象的状态信息转换为可以存储或传输的形式的过程。

serialVersionUID如何产生以及为什么在类中加上这个变量

- 当我们一个实体类中没有显示的定义一个名为“serialVersionUID”、类型为long的变量时，Java序列化机制会根据编译时的class自动生成一个serialVersionUID作为序列化版本比较，这种情况下，只有同一次编译生成的class才会生成相同的serialVersionUID。
- 譬如，当我们编写一个类时，随着时间的推移，我们因为需求改动，需要在本地类中添加其他的字段，这个时候再反序列化时便会出现serialVersionUID不一致，导致反序列化失败。那么如何解决呢？
- 解决办法为在本地类中添加一个“serialVersionUID”变量，值保持不变，便可以进行序列化和反序列化。

==、equal() 和 hashCode() 的联系和区别

- `==`：判断的是两个对象是否是同一个对象，即实际比较他们的内存地址是否指向同一个对象
（基本类型比较的是值是否相等，引用数据类型比较的是内存地址）
- `equal()`：方法存在于Object类中，所有类都可以重写这个方法，如果不重写实际效果比较的也是两个对象是否是同一个对象，与`==`效果一样
- 所以当我们想比较两个实例对象的内容相同（或者部分内容相同）即返回true而不是比较内存地址是否一样的时候，我们则可以重写`equals`方法
- `String`类就重写了`equals`方法，使得当字符串内容相同的时候就返回true

II. equals() 和 hashCode() 的联系

- `hashCode()` 方法返回的是一个int类型的哈希码，也称为散列码，哈希码的作用是确定该对象在哈希表中的索引位置，方便快速查找该对象
- `hashCode()` 方法也在Object中，当我们需要将某个类放到散列表(HashSet, Hashtable, HashMap等等)中的时候，那么此时哈希码就起到了作用
- 在往散列表中添加数据的时候：
 - 第一步：先会获取该对象的哈希码和散列表中已存在对象的哈希码进行比较，如果哈希码不相同则认为数据不重复直接添加，如果相同则执行第二步
 - 第二步：如果哈希码相同那么才会调用 `equals()` 方法来比较两个对象是否相同
 - 注意：两个对象哈希码不相同代表两个对象一定不是同一个对象或者值不相同，两个对象哈希码相同却不能判断他们两个对象或者值相同
- 所以由上面我们可以知道：当我们只重写了 `equals()` 方法的时候，每次 `hashCode()` 方法都会通过对象的内存地址 来计算哈希码导致只要对象内存地址不一样都判断为true，那么就不会再调用 `equals()` 方法来进行判断

III. 结论

综上，当我们要将类放入散列表中的时候，如果想通过比较类的内容或者部分内容来判断两个对象是否相同的时候，不仅要重写 `equals()` 方法也要重写 `hashCode()` 方法

Java 语言有哪些特点

1. 简单易学；
2. 面向对象（封装，继承，多态）；
3. 平台无关性（Java 虚拟机实现平台无关性）；
4. GC实现垃圾回收；
5. 异常处理机制；
6. 支持多线程；
7. 支持网络编程并且很方便；
8. 编译与解释并存；

List , Set , Map 的大致区别

- **List**
 - **ArrayList**: 线程不安全，底层是Object[]数组，适用于修改和查找次数多的场景
 - **LinkedList**: 线程不安全，底层是双向链表(JDK1.6之前为循环链表，JDK1.7取消了循环)，适合于增加和删除多的场景
 - **Vector**: 线程安全，底层是Object[]数组
- **Set**
 - **HashSet**: 存储元素无序且唯一，基于HashMap实现的，底层采用HashMap来保存元素
 - **LinkedHashSet**: LinkedHashSet是HashSet的子类，并且其内部是通过LinkedHashMap来实现的。
 - **TreeSet**: 存储的元素有序，红黑树(自平衡的排序二叉树)
- **Map**
 - **HashMap**: JDK1.8之前HashMap由数组+链表组成的，数组是HashMap的主体，JDK1.8以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）（将链表转换成红黑树前会判断，如果当前数组的长度小于64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间
LinkedHashMap: LinkedHashMap继承自HashMap，所以它的底层仍然是基于链式散列结构即由数组和链表或红黑树组成。另外，LinkedHashMap在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。详细可以查看：
[《LinkedHashMap源码详细分析 \(JDK1.8\)》](#)
 - **Hashtable**: 数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，线程安全的，只不过基本不采用这个，而是用CurrentHashMap，HashMap可以存储null的key和value，但null作为键只能有一个，null作为值可以有多个；HashTable不允许有null键和null值，否则会抛出NullPointerException。
 - **TreeMap**: 可以对集合中的元素根据键排序的，相比于HashMap来说 TreeMap主要多了对集合中的元素根据键排序的能力以及对集合内元素的搜索的能力，默认是按key的升序排序，不过我们也可以指定排序的比较器。
示例代码如下：

Map练习题

6. 下面的代码输出什么? Homework06.java 5min

//老韩提示：这道题很有意思，稍不注意就掉进陷阱。

已知: Person类按照id和name重写了hashCode和equals方法，问下面代码输出什么？

```
HashSet set = new HashSet(); //ok
Person p1 = new Person(1001,"AA"); //ok
Person p2 = new Person(1002,"BB"); //ok
set.add(p1); //ok
set.add(p2); //ok
p1.name = "CC";
set.remove(p1);
System.out.println(set);
set.add(new Person(1001,"CC"));
System.out.println(set);
set.add(new Person(1001,"AA"));
System.out.println(set);
```

```
HashSet set = new HashSet(); //ok
Person p1 = new Person(1001,"AA"); //ok
Person p2 = new Person(1002,"BB"); //ok
set.add(p1); //ok
set.add(p2); //ok
p1.name = "CC";
set.remove(p1);
System.out.println(set); ②
set.add(new Person(1001,"CC"));
System.out.println(set); ③
set.add(new Person(1001,"AA"));
System.out.println(set); ④
```

- **第一次输出**: 输出P1和P2，因为remove(p1)方法没有删除到p1，原因是p1的name字段已经被修改了，remove方法是根据哈希码计算索引位置删除该索引位置的对象，但是Person类的hashCode方法已经被重写为根据id和name计算，所以删除时候计算的哈希码与原先存储时候的哈希码不一样了，删除到其他索引位置去了，并没有删除p1
- **第二次输出**: 输出三个对象，因为第三次添加的时候，计算出的索引位置并没有存放其他数据，跟它现在name和id相同的p1位置是根据修改前的name计算出的索引来存储的，所以添加成功
- **第三次输出**: 输出四个对象，因为第四次添加的时候，计算出索引位置和p1相同，但是调用equals方法去比较的时候，name不相同，于是链接到了p1的后面，添加成功

事务的ACID和数据库三范式

数据三范式

- **第一范式**：强调的是列的原子性，即数据库表的每一列都是不可分割的原子数据项。
- **第二范式**：要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性。
- **第三范式**：任何非主属性不依赖于其它非主属性。

事务的ACID

- **原子性** (Atomicity)：事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
- **一致性** (Consistency)：执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的；
- **隔离性** (Isolation)：并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；
- **持久性** (Durability)：一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

事务的四种隔离级别

四种隔离级别

- **READ-UNCOMMITTED**：读未提交，最低隔离级别、事务未提交前，就可被其他事务读取（会出现幻读、脏读、不可重复读）。
- **READ-COMMITTED**：读已提交，一个事务提交后才能被其他事务读取到（会造成幻读、不可重复读）。
- **REPEATABLE-READ**：可重复读，mysql 默认级别，保证多次读取同一个数据时，其值都和事务开始时候的内容是一致，禁止读取到别的事务未提交的数据（会造成幻读）。
- **SERIALIZABLE**：序列化，代价最高最可靠的隔离级别，该隔离级别能防止脏读、不可重复读、幻读。
- 关于不同隔离级别会出现的问题
 - **脏读**：表示一个事务能够读取另一个事务中还未提交的数据。比如，某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。
 - **不可重复读**：是指在一个事务内，多次读同一数据数据发生了变化。
 - **幻读**：指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了。

MyISAM和InnoDB区别

MyISAM和InnoDB区别

MyISAM是MySQL的默认数据库引擎（5.5版之前）。虽然性能极佳，而且提供了大量的特性，包括全文索引、压缩、空间函数等，但MyISAM不支持事务和行级锁，而且最大的缺陷就是崩溃后无法安全恢复。不过，5.5版本之后，MySQL引入了InnoDB（事务性数据库引擎），MySQL 5.5版本后默认的存储引擎为InnoDB。

大多数时候我们使用的都是InnoDB存储引擎，但是在某些情况下使用MyISAM也是合适的比如读密集的情况下。（如果不介意MyISAM崩溃恢复问题的话）。

两者的对比：

- **是否支持行级锁**: MyISAM只有表级锁(table-level locking)，而InnoDB支持行级锁(row-level locking)和表级锁，默认为行级锁。
- **是否支持事务和崩溃后的安全恢复**: MyISAM强调的是性能，每次查询具有原子性，其执行速度比InnoDB类型更快，但是不提供事务支持。但是InnoDB提供事务支持，外部键等高级数据库功能。具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。
- **是否支持外键**: MyISAM不支持，而InnoDB支持。

乐观锁和悲观锁，以及乐观锁的实现方式

• 悲观锁,乐观锁

- **悲观锁**: 它对于数据被外界修改持保守态度，认为数据随时会修改。
整个数据处理中需要将数据加锁。悲观锁一般都是依靠关系数据库提供的锁机制
事实上关系数据库中的行锁，表锁不论是读写锁都是悲观锁
- **乐观锁**: 顾名思义，就是很乐观，每次自己操作数据的时候认为没有人回来修改它，所以不去加锁。但是在更新的时候会去判断在此期间数据有没有被修改
需要用户自己去实现，不会发生并发抢占资源，只有在提交操作的时候检查是否违反数据完整性

• 悲观锁,乐观锁使用前提

- 对于读操作远多于写操作的时候，这时候一个更新操作加锁会阻塞所有读取，降低了吞吐量。最后还要释放锁，锁是需要一些开销的，这时候可以选择乐观锁
如果是读写比例差距不是非常大或者你的系统没有响应不及时，吞吐量瓶颈问题，那就不要去使用乐观锁，它增加了复杂度，也带来了业务额外的风险。

• 乐观锁的实现方式

- **版本号**: 就是给数据增加一个版本标识，在数据库上就是表中增加一个version字段每次更新把这个字段加1读取数据的时候把version读出来，更新的时候比较version，如果还是开始读取的version就可以更新了如果现在的version比老的version大，说明有其他事务更新了该数据，并增加了版本号，这时候得到一个无法更新的通知，用户自行根据这个通知来决定怎么处理，比如重新开始一遍。示例
- **时间戳**: 和版本号基本一样，只是通过时间戳来判断而已，注意时间戳要使用数据库服务器的时间戳不能是业务系统的时间，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间(timestamp)和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比如果一致则OK，否则就是版本冲突。

读锁和写锁

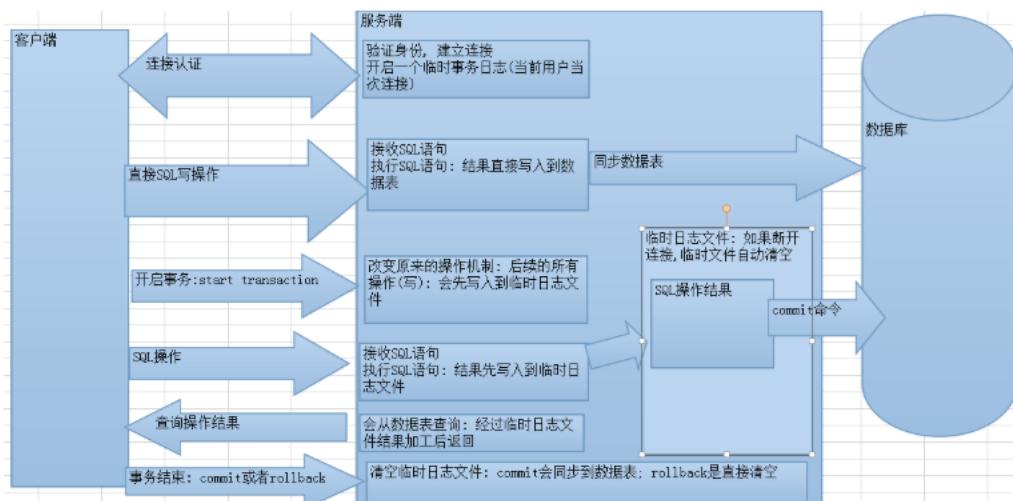
- 按操作分
 - 读锁(共享锁)**: 针对同一份数据,多个读取操作可以同时进行而不互相影响
 - 写锁(排它锁)**: 当前写操作没有完成前,会阻断其他写锁和读锁
- 按粒度分:
 - 表级锁**: MySQL中锁定粒度最大的一种锁,对当前操作的整张表加锁,实现简单,资源消耗也比较少,加锁快,不会出现死锁。其锁定粒度最大,触发锁冲突的概率最高,并发度最低, MyISAM和 InnoDB引擎都支持表级锁。
 - 行级锁**: MySQL中锁定粒度最小的一种锁,只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小,并发度高,但加锁的开销也最大,加锁慢,会出现死锁。
- 页锁
- 发生死锁的场景
 - 例如下面两条语句 第一条语句会优先使用 name 索引,因为name不是主键索引,还会用到主键索引
 - 第二条语句是首先使用主键索引,再使用name索引 如果两条语句同时执行,第一条语句执行了name索引等待第二条释放主键索引,第二条执行了主键索引等待第一条的name索引,这样就造成了死锁。

```

1  #①
2  update mk_user set name = '1' where `name`='idis12';
3  #②
4  update mk_user set name='12'  where id=12;

```

事务的原理

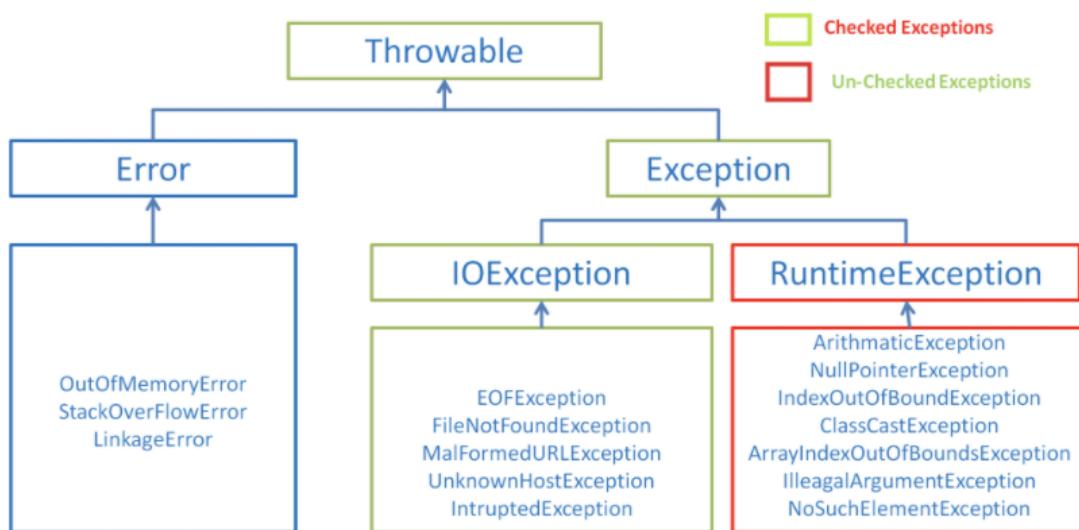


- 事务开启之后,所有的操作都会临时保存到事务日志中,事务日志只有在得到commit命令才会同步到数据表中,其他任何情况都会清空事务日志(rollback, 断开连接)
- 事务的步骤
 1. 客户端连接数据库服务器, 创建连接时创建此用户临时日志文件
 2. 开启事务以后, 所有的操作都会先写入到临时日志文件中
 3. 所有的查询操作从表中查询, 但会经过日志文件加工后才返回
 4. 如果事务提交commit则将日志文件中的数据写到表中, rollback否则清空日志文件。

数据库中char 和 varchar

- char 和 varchar
 - **char(n)** : 固定长度类型, 比如订阅 char(10), 当你输入 "abc" 三个字符的时候, 它们占的空间还是 10 个字节, 其他 7 个是空字节。
 - **char 优点** : 效率高; 缺点: 占用空间; 适用场景: 存储密码的 md5 值, 固定长度的, 使用 char 非常合适。
 - **varchar(n)** : 可变长度, 存储的值是每个值占用的字节再加上一个用来记录其长度的字节的长度。
 - 从空间上考虑 varcahr 比较合适; 从效率上考虑 char 比较合适, 二者使用需要权衡。

检查异常和非检查异常的区别



- 非检查异常 (unchecked exception) : Error 和 RuntimeException 以及他们的子类。javac在编译时, 不会提示和发现这样的异常, 不要求在程序处理这些异常, 这种异常是运行时发生, 无法预先捕捉处理的。
- 检查异常 (checked exception) : 除了Error 和 RuntimeException的其它异常。javac强制要求程序员为这样的异常做预备处理工作在方法中要么用try-catch语句捕获它并处理, 要么用throws子句声明抛出它, 否则编译不会通过。
 - 通过把IOException声明为检查型异常, Java 确保了你能够优雅的对异常进行处理。另一个可能的理由是, 可以使用catch或finally来确保数量受限的系统资源 (比如文件描述符) 在你使用后尽早得到释放。 (关于为什么Java要由检查异常的一些理由)
- 检查型异常和非检查型异常的主要区别在于其处理方式。检查型异常需要使用try, catch和finally关键字在编译期进行处理, 否则会出现编译器会报错。对于非检查型异常则不需要这样做。Java中所有继承自java.lang.Exception类的异常都是检查型异常, 所有继承自RuntimeException的异常都被称为非检查型异常。

异常处理中的finally的特性

- finally块不管异常是否发生, 只要对应的try执行了, 则它一定也执行。只有一种方法让finally块不执行: **System.exit()**。因此finally块通常用来做资源释放操作: 关闭文件, 关闭数据库连接等。
- 良好的编程习惯是: 在try块中打开资源, 在finally块中清理释放这些资源。
- 在 try块中即便有return, break, continue等改变执行流的语句, finally也会执行。

###

Mysql索引的分类和哪些情况适合建立索引

- - **单值索引**
 - 一个索引只包含单个列,一个表可以有多个单值索引,一般来说,一个表建立索引不要超过5个
 - **唯一索引**
 - 索引列的值必须唯一,但允许有空值
 - **复合索引**
 - 一个索引包含多个列
 - **全文索引**
 - MySQL全文检索是利用查询关键字和查询列内容之间的相关度进行检索,可以利用全文索引来提高匹配的速度。
- **哪些情况适合建立索引**
 - 主键自动建立唯一索引:primary
 - **频繁作为查询条件的字段应该创建索引**: 比如银行系统银行帐号,电信系统的手机号
 - **查询中与其它表关联的字段,外键关系建立索引**: 比如员工表的部门外键
 - **频繁更新的字段不适合建立索引**: 每次更新不单单更新数据,还要更新索引
 - **where条件里用不到的字段不建立索引**
 - **查询中排序的字段,排序的字段若通过索引去访问将大大提升排序速度**
 - **查询中统计或分组的字段**
 - 经常增删改的表和数据重复的表字段不适合建立索引
- **最佳左前缀法则 : 如果索引有多列,要遵守最左前缀法则,指的就是从索引的最左列开始 并且不跳过索引中的列**

MySQL如何正确使用索引

- **使用全值匹配**, 使用全部的复合索引起起查询是性能最好的, 也可以最大限度避免索引的失效
- **最佳左前缀法则** : 如果索引有多列,要遵守最左前缀法则,指的就是从索引的最左列开始 并且不跳过索引中的列
- **不在索引列上做任何操作** : 计算,函数,类型转换会导致索引失效而转向全表扫描 如: EXPLAIN

```
SELECT * FROM employee WHERE trim(NAME)='白骨精'
```
- mysql在使用不等于(!=或者<>)的时候无法使用索引会导致全表扫描
- 少用or 用or连接时, 会导致索引失效
- like以通配符开头(%)索引失效变成全表扫描

重载和重写的区别

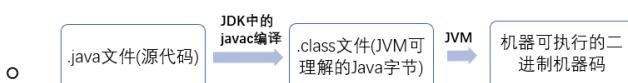
- **重载**就是同样的一个方法能够根据输入数据的不同, 做出不同的处理
 - 发生在同一个类中 (或者父类和子类之间), 方法名必须相同, 参数类型不同、个数不同、顺序不同, 方法返回值和访问修饰符可以不同。
- **重写**就是当子类继承自父类的相同方法, 输入数据一样, 但要做出有别于父类的响应时, 你就要覆盖父类方法
 - 返回值类型、方法名、参数列表必须相同, 抛出的异常范围小于等于父类, 访问修饰符范围大于等于父类。
 - 如果父类方法访问修饰符为 `private/final/static` 则子类就不能重写该方法, 但是被 `static` 修饰的方法能够被再次声明。
 - 构造方法无法被重写

区别点	重载方法	重写方法
发生范围	同一个类	子类
参数列表	必须修改	一定不能修改
返回类型	可修改	子类方法返回值类型应比父类方法返回值类型更小或相等
异常	可修改	子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相等；
访问修饰符	可修改	一定不能做更严格的限制（可以降低限制）
发生阶段	编译期	运行期

JVM JDK 和 JRE 的各自的作用和区别

- Java 虚拟机 (JVM) 是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现 (Windows, Linux, macOS)，目的是使用相同的字节码，它们都会给出相同的结果。字节码和不同系统的 JVM 实现是 Java 语言“一次编译，随处可以运行”的关键所在。

Java 程序从源代码到运行一般有下面 3 步：



我们需要格外注意的是 .class->机器码 这一步。在这一步 JVM 类加载器首先加载字节码文件，然后通过解释器逐行解释执行，这种方式的执行速度会相对比较慢。而且，有些方法和代码块是经常需要被调用的（也就是所谓的热点代码），所以后面引进了 JIT 编译器，而 JIT 属于运行时编译。当 JIT 编译器完成第一次编译后，其会将字节码对应的机器码保存下来，下次可以直接使用。而我们知道，机器码的运行效率肯定是高于 Java 解释器的。这也解释了我们为什么经常会说 Java 是编译与解释共存的语言。

- JDK 是 Java Development Kit 缩写，它是功能齐全的 Java SDK。它拥有 JRE 所拥有的一切，还有编译器 (javac) 和工具 (如 javadoc 和 jdb)。它能够创建和编译程序。**
- JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java 虚拟机 (JVM)，Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。**
- 如果你只是为了运行一下 Java 程序的话，那么你只需要安装 JRE 就可以了。如果你需要进行一些 Java 编程方面的工作，那么你就需要安装 JDK 了。但是，这不是绝对的。有时，即使您不打算在计算机上进行任何 Java 开发，仍然需要安装 JDK。例如，如果要使用 JSP 部署 Web 应用程序，那么从技术上讲，您只是在应用程序服务器中运行 Java 程序。那你为什么需要 JDK 呢？因为应用程序服务器会将 JSP 转换为 Java servlet，并且需要使用 JDK 来编译 servlet。

Java 泛型了解么？什么是类型擦除？介绍一下常用的通配符

- 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。
- Java 的泛型是伪泛型，这是因为 Java 在编译期间，所有的泛型信息都会被擦掉，这也就是通常所说类型擦除，如下面的例子，反射得到类就没有泛型的限制**

```

o List<Integer> list = new ArrayList<>();
list.add(12);
//这里直接添加会报错
list.add("a");
Class<? extends List> clazz = list.getClass();
Method add = clazz.getDeclaredMethod("add", Object.class);
//但是通过反射添加，是可以的
add.invoke(list, "k1");
System.out.println(list)

```

- 常用的通配符为： T, E, K, V, ?
- ? 表示不确定的 java 类型
- T (type) 表示具体的一个 java 类型
- K V (key value) 分别代表 java 键值中的 Key Value
- E (element) 代表 Element

Java 中的几种基本数据类型是什么？对应的包装类型是什么？各自占用多少字节？

- Java中有 8 种基本数据类型：byte、short、int、long、float、double、char、boolean
- 八种基本类型都有对应的包装类分别为：Byte、Short、Integer、Long、Float、Double、Character、Boolean

基本类型	位数	字节	默认值
int	32	4	0
short	16	2	0
long	64	8	0L
byte	8	1	0
char	16	2	'u0000'
float	32	4	0f
double	64	8	0d
boolean	1		false

- 对于 boolean，官方文档未明确定义，它依赖于 JVM 厂商的具体实现。逻辑上理解是占用 1 位，但是实际中会考虑计算机高效存储因素

构造方法有哪些特性

面向对象的三大特征

- 封装：封装是指把一个对象的状态信息（也就是属性）隐藏在对象内部，不允许外部对象直接访问对象的内部信息。但是可以提供一些可以被外界访问的方法来操作属性。
- 继承：继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承，可以快速地创建新的类，可以提高代码的重用，程序的可维护性，节省大量创建新类的时间，提高我们的开发效率。
 - 子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，**只是拥有**。
 - 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
- 多态：顾名思义，表示一个对象具有多种的状态。具体表现为父类的引用指向子类的实例。

- 对象类型和引用类型之间具有继承（类）/实现（接口）的关系；
- 引用类型变量发出的方法调用的到底是哪个类中的方法，必须在程序运行期间才能确定；
- 多态不能调用“只在子类存在但在父类不存在”的方法；
- 如果子类重写了父类的方法，真正执行的是子类覆盖的方法，如果子类没有覆盖父类的方法，执行的是父类的方法。

String StringBuffer 和 StringBuilder 的区别是什么？String 为什么是不可变的

- `String` 类中使用 `final` 关键字修饰字符数组来保存字符串，`private final char value[]`，所以 `String` 对象是不可变的
- `StringBuilder` 与 `StringBuffer` 都继承自 `AbstractStringBuilder` 类，在 `AbstractStringBuilder` 中也是使用字符数组保存字符串 `char[] value` 但是没有用 `final` 关键字修饰，所以这两种对象都是可变的
- 线程安全性：
 - `String` 中的对象是不可变的，也就可以理解为常量，线程安全
 - `StringBuffer` 对方法加了同步锁或者对调用的方法加了同步锁，线程安全
 - `StringBuilder` 并没有对方法进行加同步锁，线程不安全
- 对于三者使用的总结：
 1. 操作少量的数据：适用 `String`
 2. 单线程操作字符串缓冲区下操作大量数据：适用 `StringBuilder`
 3. 多线程操作字符串缓冲区下操作大量数据：适用 `StringBuffer`

Object 类的常见方法有哪些

`public final native Class<?> getClass()`//native方法，用于返回当前运行时对象的Class对象，使用了`final`关键字修饰，故不允许子类重写。

`public native int hashCode()` //native方法，用于返回对象的哈希码，主要使用在哈希表中，比如JDK中的`HashMap`。

`public boolean equals(Object obj)`//用于比较2个对象的内存地址是否相等，`String`类对该方法进行了重写用户比较字符串的值是否相等。

`protected native Object clone()` throws `CloneNotSupportedException`//native方法，用于创建并返回当前对象的一份拷贝。一般情况下，对于任何对象 `x`，表达式 `x.clone() != x` 为`true`，`x.clone().getClass() == x.getClass()` 为`true`。`Object`本身没有实现`Cloneable`接口，所以不重写`clone`方法并且进行调用的话会发生`CloneNotSupportedException`异常。

`public String toString()`//返回类的名字@实例的哈希码的16进制的字符串。建议`Object`所有的子类都重写这个方法。

`public final native void notify()`//native方法，并且不能重写。唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念)。如果有多个线程在等待只会任意唤醒一个。

`public final native void notifyAll()`//native方法，并且不能重写。跟`notify`一样，唯一的区别就是会唤醒在此对象监视器上等待的所有线程，而不是一个线程。

```
public final native void wait(long timeout) throws InterruptedException//native方法，并且不能重写。暂停线程的执行。注意：sleep方法没有释放锁，而wait方法释放了锁。timeout是等待时间。
```

```
public final void wait(long timeout, int nanos) throws InterruptedException//多了nanos参数，这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。所以超时的时间还需要加上nanos毫秒。
```

```
public final void wait() throws InterruptedException//跟之前的2个wait方法一样，只不过该方法一直等待，没有超时时间这个概念
```

```
protected void finalize() throws Throwable {}//实例被垃圾回收器回收的时候触发的操作
```

简述线程、程序、进程的基本概念。以及他们之间关系是什么？

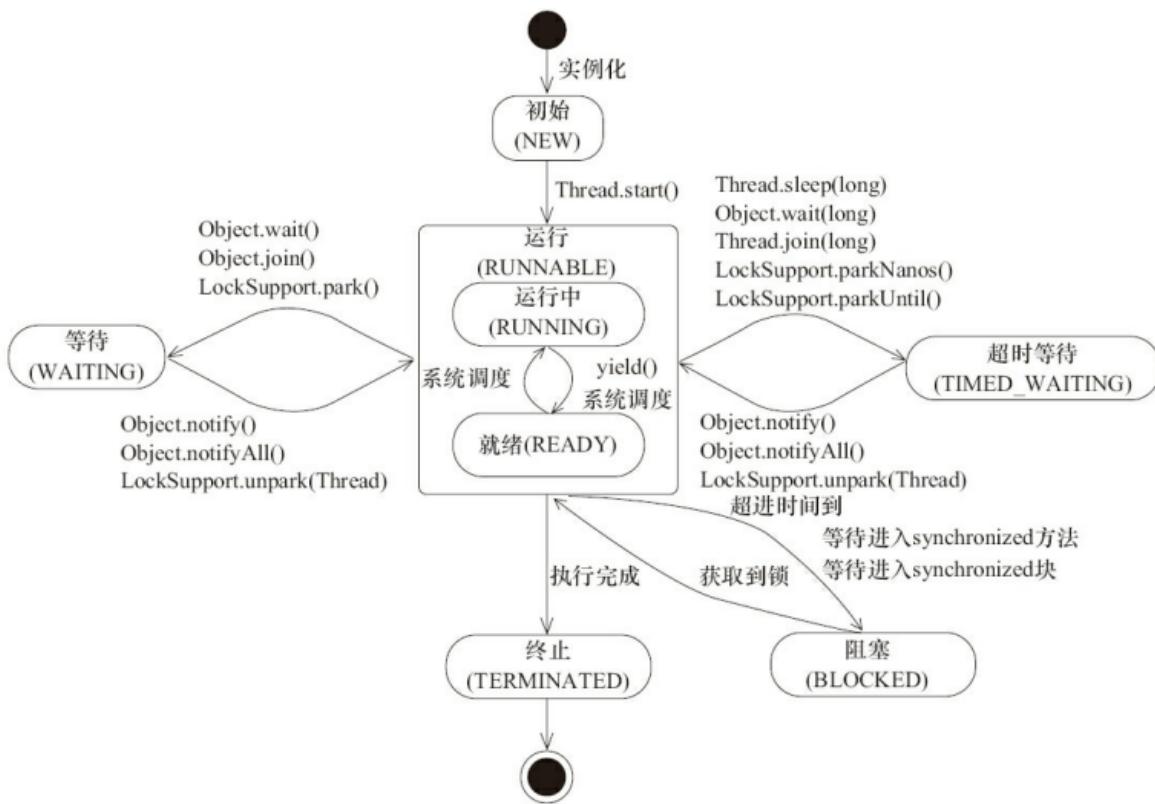
- **线程**与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。
- **程序**是含有指令和数据的文件，被存储在磁盘或其他的数据存储设备中，也就是说程序是静态的代码。
- **进程**是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如 CPU 时间，内存空间，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将被操作系统载入内存中。线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

线程有哪些基本状态？

Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态（图源《Java 并发编程艺术》4.1.4 节）

状态名称	说 明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状态变迁如下图所示（图源《Java 并发编程艺术》4.1.4 节）：



BIO,NIO,AIO 有什么区别?(这个还不熟悉)

- **BIO (Blocking I/O):** 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机 1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。
- **NIO (Non-blocking/New I/O):** NIO 是一种同步非阻塞的 I/O 模型，在 Java 1.4 中引入了 NIO 框架，对应 `java.nio` 包，提供了 `Channel`, `Selector`, `Buffer` 等抽象。NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的 I/O 操作方法。NIO 提供了与传统 BIO 模型中的 `socket` 和 `ServerSocket` 相对应的 `socketChannel` 和 `ServerSocketChannel` 两种不同的套接字通道实现，两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞 I/O 来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发
- **AIO (Asynchronous I/O):** AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2，它是异步非阻塞的 IO 模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步 IO 的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO 操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

引用类型的大致创建过程

现在为其创建一个对象 Student d1= new Student (8,8,2021);

在内存中的具体创建过程是：

- 1) 首先在栈内存中为d1分配一块空间；
- 2) 然后在堆内存中为Student 对象分配一块空间，并为其三个属性设初值0, 0, 0；
- 3) 根据类Student 中对属性的定义，为该对象的三个属性进行赋值操作；
- 4) 调用构造方法，为三个属性赋值为8, 8, 2021；（注意这个时候d1与Student 对象之间还没有建立联系）
- 5) 将Student 对象在堆内存中的地址，赋值给栈中的d1;通过句柄d1可以找到堆中对象的具体信息。

final 在 java 中有什么作用

- final 修饰的类叫最终类，该类不能被继承。
- final 修饰的方法不能被重写。
- final 修饰的变量不可更改，其不可更改指的是其引用不可修改，对于引用类型值还是可能改变的，
举个例子：String 内部对于 value 的定义；而对于基本类型来说就叫做常量了。
- final、finally、finalize 有什么区别？
 - final可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。
 - finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执行的代码方法finally代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。
 - finalize是一个方法，属于Object类的一个方法，而Object类是所有类的父类，该方法一般由垃圾回收器来调用，当我们调用System的gc()方法的时候，由垃圾回收器调用finalize(),回收垃圾。

String 为什么设置为不可变？

- 为了实现字符串常量池(如果字符串是可变的，某一个字符串变量改变了其值，那么其指向的变量的值也会改变，String pool将不能够实现)
- 为了线程安全(字符串自己便是线程安全的)
- 可以加快字符串处理速度，保证同一个对象调用 hashCode() 都产生相同的值，于是在创建对象时其hashcode就可以放心的缓存，不需要重新计算。这也是 Map 类的 key 使用 String 的原因。

接口和抽象类的区别

- 接口中不能有非抽象方法，而抽象方法中可以有非抽象方法
- 实现接口的关键字为implements，继承抽象类的关键字为extends。一个类可以实现多个接口，但一个类只能继承一个抽象类。所以，使用接口可以间接地实现多重继承
- 接口成员变量默认为public static final，必须赋初值，不能被修改，其所有的成员方法都是public、abstract的，抽象方法可以有public、protected这些修饰符

有关字符串常量池

(JVM每个内存区域以及一些常量池的知识点整理下)

###

琐碎的知识

- new 创建对象实例（对象实例在堆内存中），对象引用指向对象实例（对象引用存放在栈内存中）。一个对象引用可以指向 0 个或 1 个对象（一根绳子可以不系气球，也可以系一个气球）；一个对象可以有 n 个引用指向它（可以用 n 条绳子系住一个气球）
- 构造方法的特性：
 - 名字与类名相同
 - 没有返回值，但不能用 void 声明构造函数
- 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
- 在一个静态方法内调用一个非静态成员为什么是非法的：静态方法在类加载的时候就会分配内存，非静态成员(变量或方法)属于类的对象，只有在类的对象产生(实例化)时才会分配内存，然后通过类的对象(实例)去访问
- **java中的基本数据类型存放位置：**在方法中声明的基本类型变量存放在方法栈中，随着方法栈出栈而失效，在类中声明的基本类型变量其变量名及其值放在堆内存中
-

集合面试题Anki

说说 List,Set,Map 三者的区别

- `List`(对付顺序的好帮手): 存储的元素是有序的、可重复的。
- `Set`(注重独一无二的性质): 存储的元素是无序的、不可重复的。
- `Map`(用 Key 来搜索的专家): 使用键值对 (key-value) 存储, 类似于数学上的函数 $y=f(x)$, "x" 代表 key, "y" 代表 value, Key 是无序的、不可重复的, value 是无序的、可重复的, 每个键最多映射到一个值。

1.1.3.1. List

- `ArrayList` : `Object[]` 数组
- `Vector` : `Object[]` 数组
- `LinkedList` : 双向链表(JDK1.6 之前为循环链表, JDK1.7 取消了循环)

1.1.3.2. Set

- `HashSet` (无序, 唯一) : 基于 `HashMap` 实现的, 底层采用 `HashMap` 来保存元素
- `LinkedHashSet` : `LinkedHashSet` 是 `HashSet` 的子类, 并且其内部是通过 `LinkedHashMap` 来实现的。有点类似于我们之前说的 `LinkedHashMap` 其内部是基于 `HashMap` 实现一样, 不过还是有一点点区别的
- `TreeSet` (有序, 唯一) : 红黑树(自平衡的排序二叉树)

为什么要使用集合?

当我们需要保存一组类型相同的数据的时候, 我们应该是用一个容器来保存, 这个容器就是数组, 但是, 使用数组存储对象具有一定的弊端, 因为我们在实际开发中, 存储的数据的类型是多种多样的, 于是, 就出现了“集合”, 集合同样也是用来存储多个数据的。

数组的缺点是一旦声明之后, 长度就不可变了; 同时, 声明数组时的数据类型也决定了该数组存储的数据的类型; 而且, 数组存储的数据是有序的、可重复的, 特点单一。但是集合提高了数据存储的灵活性, Java 集合不仅可以用来存储不同类型不同数量的对象, 还可以保存具有映射关系的数据。

RandomAccess 接口的作用

- 查看源码我们发现实际上 `RandomAccess` 接口中什么都没有定义。所以, 在我看来 `RandomAccess` 接口不过是一个标识罢了。标识什么? 标识实现这个接口的类具有随机访问功能。
- 判断传入的 list 是否 `RandomAccess` 的实例, 如果是, 调用 `indexedBinarySearch()` 方法, 如果不是, 那么调用 `iteratorBinarySearch()` 方法
- 数组天然支持随机访问, 时间复杂度为 $O(1)$, 所以称为快速随机访问。链表需要遍历到特定位置才能访问特定位置的元素, 时间复杂度为 $O(n)$, 所以不支持快速随机访问。
- `ArrayList` 实现了 `RandomAccess` 接口, 就表明了他具有快速随机访问功能。`RandomAccess` 接口只是标识, 并不是说 `ArrayList` 实现 `RandomAccess` 接口才具有快速随机访问功能的

说一说 ArrayList 的扩容机制

- **自己的跟踪源码的理解：**arrayList 使用无参构造方法创建list对象后，实际上初始化赋值是一个长度为0的空数组，然后第一次调用add方法添加数据的时候，进去扩容方法的时候，首先会判断此时的数组是不是这个空数组，如果是则会返回默认大小10去触发扩容机制，如果不是才直接返回此时数组的size大小，当返回10的时候，经过一次判断然后调用Arrays.copyOf(elementData, newCapacity)数组扩容到10，此时数组返回一个大小为10的新数组，下次再次添加的时候就不会进入第一次添加的那个判断中去
- **以无参数构造方法创建 ArrayList 时，实际上初始化赋值的是一个空数组。当真正对数组进行添加元素操作时，才真正分配容量。即向数组中添加第一个元素时，数组容量扩为 10**

说下集合中的无序性和不可重复性的含义是什么

- 什么是无序性？无序性不等于随机性，无序性是指存储的数据在底层数组中并非按照数组索引的顺序添加，而是根据数据的哈希值决定的。
- 什么是不可重复性？不可重复性是指添加的元素按照 equals() 判断时，返回 false，需要同时重写 equals() 方法和 hashCode() 方法。

HashSet、LinkedHashSet 和 TreeSet 三者的异同

- HashSet 的底层是 HashMap，线程不安全的，可以存储 null 值，根据 hashCode 值来决定该对象在 HashSet 中存储位置
- LinkedHashSet 是 HashSet 的子类，能够按照添加的顺序遍历
- TreeSet 底层使用红黑树，可以确保集合元素处于排序状态。TreeSet 支持两种排序方式，自然排序和定制排序，其中自然排序为默认的排序方式。

ArrayList 和 Vector LinkedList 的区别

- arrayList 是 List 的主要实现类，底层使用 Object[] 存储，适用于频繁的查找工作，线程不安全
- Vector，底层使用 Object[] 存储，线程安全的。
- LinkedList 底层使用的是 双向链表 数据结构

HashMap 和 Hashtable 的区别

1.4.1. HashMap 和 Hashtable 的区别

1. 线程是否安全：HashMap 是非线程安全的，Hashtable 是线程安全的，因为 Hashtable 内部的方法基本都经过 synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；
2. 效率：因为线程安全的问题，HashMap 要比 Hashtable 效率高一点。另外，Hashtable 基本被淘汰，不要在代码中使用它；
3. 对 Null key 和 Null value 的支持：HashMap 可以存储 null 的 key 和 value，但 null 作为键只能有一个，null 作为值可以有多个；Hashtable 不允许有 null 键和 null 值，否则会抛出 NullPointerException。
4. 初始容量大小和每次扩充容量大小的不同：① 创建时如果不指定容量初始值，Hashtable 默认的初始大小为 11，之后每次扩充，容量变为原来的 2n+1，HashMap 默认的初始化大小为 16，之后每次扩充，容量变为原来的 2 倍。② 创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为 2 的幂次方大小（HashMap 中的 tableSizeFor() 方法保证，下面给出了源代码）。也就是说 HashMap 总是使用 2 的幂作为哈希表的大小，后面会介绍到为什么是 2 的幂次方。
5. 底层数据结构：JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

HashMap 和 HashSet 区别

如果你看过 `HashSet` 源码的话就应该知道：`HashSet` 底层就是基于 `HashMap` 实现的。（`HashSet` 的源码非常非常少，因为除了 `clone()`、`writeObject()`、`readObject()` 是 `HashSet` 自己不得不实现之外，其他方法都是直接调用 `HashMap` 中的方法。）

HashMap	HashSet
实现了 <code>Map</code> 接口	实现 <code>Set</code> 接口
存储键值对	仅存储对象
调用 <code>put()</code> 向 map 中添加元素	调用 <code>add()</code> 方法向 <code>set</code> 中添加元素
<code>HashMap</code> 使用键（Key）计算 <code>hashCode</code>	<code>HashSet</code> 使用成员对象来计算 <code>hashCode</code> 值，对于两个对象来说 <code>hashCode</code> 可能相同，所以 <code>equals()</code> 方法用来判断对象的相等性

HashSet 如何检查重复

当你把对象加入 `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` 就不会让加入操作成功。

- `hashCode()` 与 `equals()` 的相关规定：

1. 如果两个对象相等，则 `hashCode` 一定也是相同的
2. 两个对象相等，对两个 `equals()` 方法返回 `true`
3. 两个对象有相同的 `hashCode` 值，它们也不一定是相等的
4. 综上，`equals()` 方法被覆盖过，则 `hashCode()` 方法也必须被覆盖
5. `hashCode()` 的默认行为是对堆上的对象产生独特值。如果没有重写 `hashCode()`，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

- == 与 `equals` 的区别

- 对于基本类型来说，`==` 比较的是值是否相等； -
- 对于引用类型来说，`==` 比较的是两个引用是否指向同一个对象地址（两者在内存中存放的地址（堆内存地址）是否指向同一个地方）；
- 对于引用类型（包括包装类型）来说，`equals` 如果没有被重写，对比它们的地址是否相等；如果 `equals()` 方法被重写（例如 `String`），则比较的是地址里的内容。

说一下HashMap的底层实现

- JDK1.8 之前 `HashMap` 底层是 数组和链表 结合在一起使用也就是 链表散列。`HashMap` 通过 `key` 的 `hashCode` 经过扰动函数处理过后得到 `hash` 值，然后通过 $(n - 1) \& hash$ 判断当前元素存放的位置（这里的 `n` 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 `hash` 值以及 `key` 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。

所谓扰动函数指的就是 `HashMap` 的 `hash` 方法。使用 `hash` 方法也就是扰动函数是为了防止一些实现比较差的 `hashCode()` 方法。换句话说使用扰动函数之后可以减少碰撞。

- JDK1.8 之后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）（将链表转换成红黑树前会判断，如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树）时，将链表转化为红黑树，以减少搜索时间。

HashMap 的长度为什么是2的幂次方？

为了能让 `HashMap` 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，`Hash` 值的范围值-2147483648 到2147483647，前后加起来大概40亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个40亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“`(n - 1) & hash`”。（`n`代表数组长度）。这也就解释了 `HashMap` 的长度为什么是2的幂次方。

这个算法应该如何设计呢？

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：“取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说 `hash%length==hash&(length-1)` 的前提是 `length` 是2的 n 次方；）。”并且 采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 `HashMap` 的长度为什么是2的幂次方。

那么 `a % b` 操作为什么等于 `a & (b - 1)` 呢？（前提是 `b` 等于2的n次幂）

举例说明：

若 `a = 10, b = 8`, 10与8取余应得2.

8的二进制为: 1000 ; 7的二进制为: 0111.

也就是说——2的n次幂减一这样的数的二进制都是如0000111111这样前半部分是0后半部分是1的形式。

所以, 用2的n次幂减一这样的数 & 另一个数就相当于 这个数取余 (%) 2的n次幂

ConcurrentHashMap 和 Hashtable 的区别

`ConcurrentHashMap` 和 `Hashtable` 的区别主要体现在实现线程安全的方式上不同。

- **底层数据结构：**JDK1.7 的 `ConcurrentHashMap` 底层采用 **分段的数组+链表** 实现，JDK1.8 采用的数据结构跟 `HashMap` 的结构一样，数组+链表/红黑二叉树。`Hashtable` 和 JDK1.8 之前的 `HashMap` 的底层数据结构类似都是采用 **数组+链表** 的形式，数组是 `HashMap` 的主体，链表则是主要为了解决哈希冲突而存在的；
- **实现线程安全的方式（重要）：**
 - ① 在 **JDK1.7** 的时候，`ConcurrentHashMap` (**分段锁**) 对整个桶数组进行了分割分段 (`Segment`)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。
 - 到了 **JDK1.8** 的时候已经摒弃了 `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。（**JDK1.6** 以后对 `synchronized` 锁做了很多优化）整个看起来就像是优化过且线程安全的 `HashMap`，虽然在 **JDK1.8** 中还能看到 `Segment` 的数据结构，但是已经简化了属性，只是为了兼容旧版本；
 - ② `Hashtable` (**同一把锁**) : 使用 `synchronized` 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 `put` 添加元素，另一个线程不能使用 `put` 添加元素，也不能使用 `get`，竞争会越来越激烈效率越低

ConcurrentHashMap 线程安全的具体实现方式

- JDK1.7

ConcurrentHashMap 维护了一个 Segment 数组，Segment 这个类继承了重入锁 ReentrantLock，在该类里面维护了一个 HashEntry<K,V>[] table 数组，在写操作 put, remove, 扩容的时候，会对 Segment 加锁，所以仅仅影响这个 Segment，不同的 Segment 还是可以并发的，所以解决了线程的安全问题，同时又采用了分段锁也提升了并发的效率

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。

Segment 实现了 ReentrantLock，所以 Segment 是一种可重入锁，扮演锁的角色。HashEntry 用于存储键值对数据。

```
static class Segment<K,V> extends ReentrantLock implements Serializable {  
}
```

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和 HashMap 类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护着一个 HashEntry 数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 的锁。

- JDK1.8

ConcurrentHashMap 取消了 Segment 分段锁，采用 CAS 和 synchronized 来保证并发安全。数据结构跟 HashMap1.8 的结构类似，数组+链表/红黑二叉树。Java 8 在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为 O(N)）转换为红黑树（寻址时间复杂度为 O(log(N))）

synchronized 只锁定当前链表或红黑二叉树的首节点，这样只要 hash 不冲突，就不会产生并发，效率又提升 N 倍。

Set 集合怎么实现线程安全？

方案一：

和 list 一样，使用 Collections 这个工具类 syn 方法类创建个线程安全的 set.

```
Set synSet = Collections.synchronizedSet(new HashSet<>());
```

方案二：

使用 JUC 包里面的 CopyOnWriteArrayList

```
Set copySet = new CopyOnWriteArrayList<>();
```

Collections 工具类常用方法

1.5.1. 排序操作

```
void reverse(List list)//反转  
void shuffle(List list)//随机排序  
void sort(List list)//按自然排序的升序排序  
void sort(List list, Comparator c)//定制排序，由Comparator控制排序逻辑  
void swap(List list, int i, int j)//交换两个索引位置的元素  
void rotate(List list, int distance)//旋转。当distance为正数时，将list后distance个元素整体移到前面。当distance为负数时，将 list的前distan
```

1.5.2. 查找,替换操作

```
int binarySearch(List list, Object key)//对List进行二分查找，返回索引，注意List必须是有序的  
int max(Collection coll)//根据元素的自然顺序，返回最大的元素。类比int min(Collection coll)  
int max(Collection coll, Comparator c)//根据定制排序，返回最大元素，排序规则由Comparator类控制。类比int min(Collection coll, Comparator  
void fill(List list, Object obj)//用指定的元素代替指定list中的所有元素。  
int frequency(Collection c, Object o)//统计元素出现次数  
int indexOfSubList(List list, List target)//统计target在list中第一次出现的索引，找不到则返回-1，类比int lastIndexOfSubList(List source,  
boolean replaceAll(List list, Object oldVal, Object newVal), 用新元素替换旧元素
```

1.5.3. 同步控制

Collections 提供了多个 `synchronizedXxx()` 方法，该方法可以将指定集合包装成线程同步的集合，从而解决多线程并发访问集合时的线程安全问题。

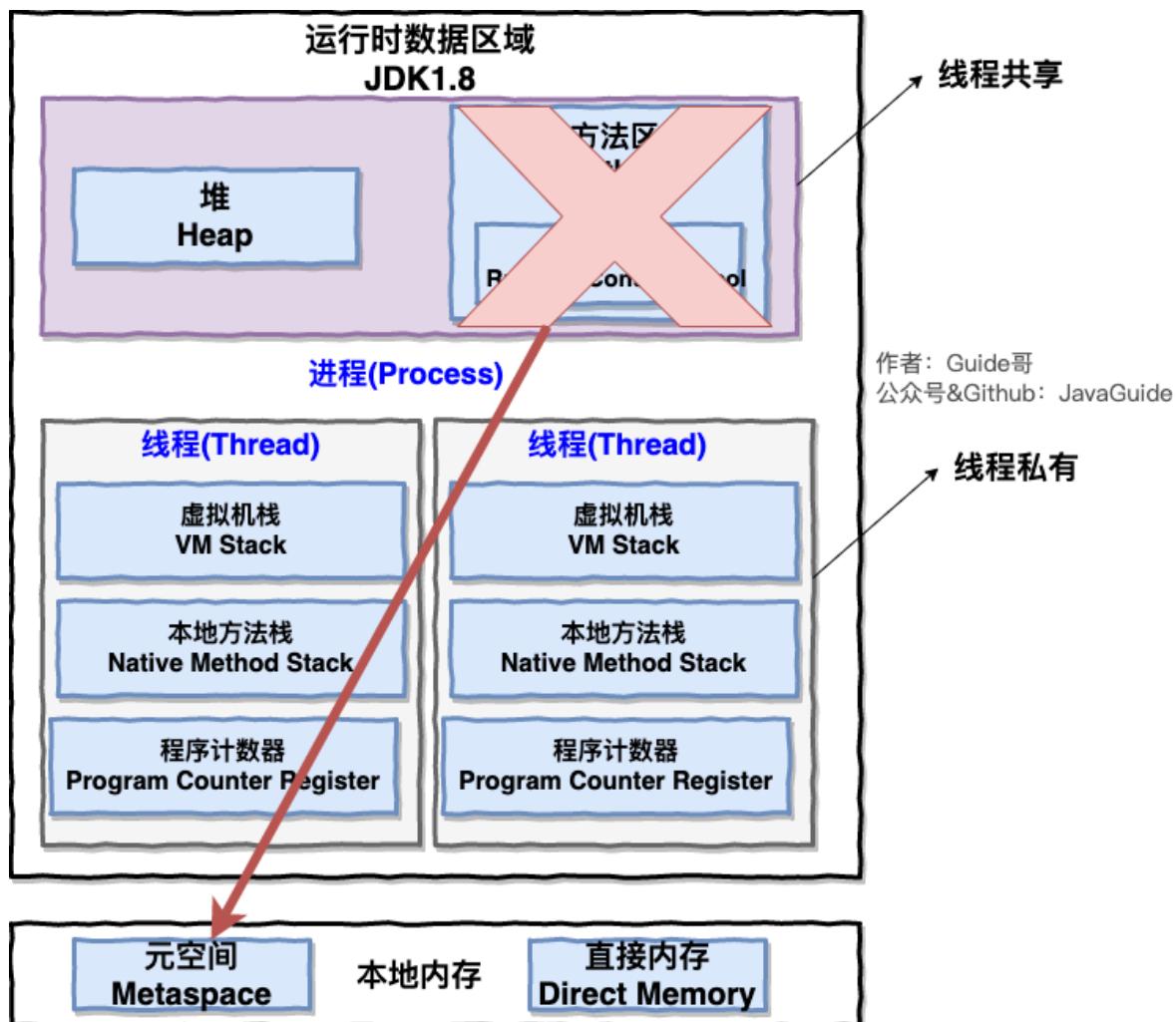
我们知道 `HashSet`, `TreeSet`, `ArrayList`, `LinkedList`, `HashMap`, `TreeMap` 都是线程不安全的。Collections 提供了多个静态方法可以把他们包装成线程同步的集合。

琐碎速记

- `arrayList` 使用无参构造方法创建list对象后，实际上初始化赋值是一个长度为0的空数组，然后第一次添加数据的时候会判断此时是不是这个空数组，如果是则会返回默认大小10去触发扩容机制，然后调用`Arrays.copyOf(elementData, newCapacity)`数组扩容到10，此时数组返回一个新数组，下次再次添加的时候就不会进入第一次添加的那个判断中去
-

JVM面试题Anki

介绍下 Java 内存区域，说出每个区域保存什么数据（分别哪些是线程私有和共享的）



- **程序计数器**: 程序计数器是一块较小的内存空间，是当前线程正在执行的那条字节码指令的地址
 - 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制
 - 在多线程情况下，程序计数器记录的是当前线程执行的位置，从而当线程切换回来时，就知道上次线程执行到哪了
 - 程序计数器是唯一一个不会出现 `OutOfMemoryError` 的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。
- **虚拟机栈**: Java 虚拟机栈是描述 Java 方法运行过程的内存模型
 - Java 虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息
 - Java 虚拟机栈会出现两种错误：`StackOverflowError` 和 `OutOfMemoryError`
 - `StackOverflowError`: 若 Java 虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前 Java 虚拟机栈的最大深度的时候，就抛出 `StackOverflowError` 错误。
 - `OutOfMemoryError`: Java 虚拟机栈的内存大小可以动态扩展，如果虚拟机在动态扩展栈时无法申请到足够的内存空间，则抛出 `OutOfMemoryError` 异常

- **本地方法栈**: 和虚拟机栈所发挥的作用非常相似，区别是：**虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务**。在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。
- **堆**: Java 虚拟机所管理的内存中最大的一块，Java 堆是所有线程共享的一块内存区域，在虚拟机启动时创建。**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组都在这里分配内存**。
 - 由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：**新生代和老年代**：再细致一点有：Eden 空间、From Survivor、To Survivor 空间等
- **方法区**: 方法区与 Java 堆一样，是各个线程共享的内存区域，它用于存储**已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据**
 - **JDK1.8 hotspot移除了永久代用元空间(Metaspace)取而代之，这时候字符串常量池还在堆，运行时常量池还在方法区**，只不过方法区的实现从永久代变成了元空间(Metaspace)

字符串常量池、Class常量池和运行时常量池的区别

- **字符串常量池**: 也叫全局字符串池，**字符串常量池里的内容是在类加载完成，经过验证，准备阶段之后在堆中生成字符串对象实例，然后将该字符串对象实例的引用值存到string pool中**（记住：**string pool中存的是引用值而不是具体的实例对象，具体的实例对象是在堆中开辟的一块空间存放的**）。这个StringTable在每个HotSpot VM的实例只有一份，被所有的类共享。
- **Class文件常量池**: class文件中除了包含类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池(constant pool table)，用于存放编译器生成的各种字面量(Literal)和符号引用(Symbolic References)。
- **运行时常量池**: 当java文件被编译成class文件之后，也就是会生成我上面所说的class常量池，jvm在执行某个类的时候，必须经过加载、连接、初始化，而连接又包括验证、准备、解析三个阶段。而**当类加载到内存中后，jvm就会将class常量池中的内容存放到运行时常量池中**，由此可知，运行时常量池也是每个类都有一个。经过解析(resolve)之后，也就是把符号引用替换为直接引用，解析的过程会去查询全局字符串池，也就是我们上面所说的StringTable，以保证运行时常量池所引用的字符串与全局字符串池中所引用的是一致的。
- 三个常量池的各个运行阶段：在该类的class常量池中存放一些符号引用，然后**类加载之后，将class常量池中存放的符号引用转存到运行时常量池中**，然后**经过验证，准备阶段之后，在堆中生成驻留字符串的实例对象**（也就是上例中str1所指向的“abc”实例对象），然后将这个对象的引用存到全局String Pool中，也就是StringTable中，最后在**解析阶段**，要把运行时常量池中的符号引用替换成直接引用，那么就直接查询StringTable，保证StringTable里的引用值与运行时常量池中的引用值一致，大概整个过程就是这样了。
- **Class常量池和运行时常量池在方法区中也就是HotSpot中的元空间里面，而字符串常量池在JDK8之后被放在了堆中**
- **总结**：
 - 1.字符串常量池在每个VM中只有一份，存放的是字符串常量的引用值。
 - 2.Class常量池是在编译的时候每个class都有的，在编译阶段，存放的是常量的符号引用。
 - 3.运行时常量池是在类加载完成之后，将每个class常量池中的符号引用值转存到运行时常量池中，也就是说，每个class都有一个运行时常量池，类在解析之后，将符号引用替换成直接引用，与全局常量池中的引用值保持一致。
-



- 上图和大部分信息来源：https://blog.csdn.net/qq_26222859/article/details/73135660

对象的创建过程

下图便是 Java 对象的创建过程，我建议最好是能默写出来，并且要掌握每一步在做什么。



Step1:类加载检查

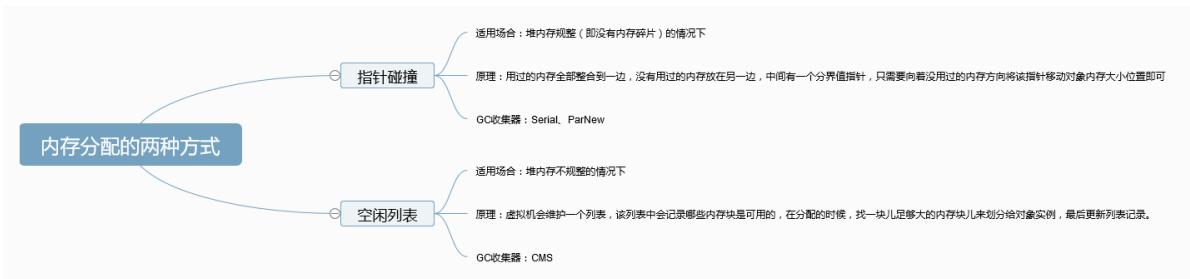
虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

Step2:分配内存

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配方式有“指针碰撞”和“空闲列表”两种，选择哪种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

内存分配的两种方式：(补充内容，需要掌握)

选择以上两种方式中的哪一种，取决于 Java 堆内存是否规整。而 Java 堆内存是否规整，取决于 GC 收集器的算法是“标记-清除”，还是“标记-整理”（也称作“标记-压缩”），值得注意的是，复制算法内存也是规整的。



内存分配并发问题（补充内容，需要掌握）

在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：

- **CAS+失败重试：** CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。**虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。**
- **TLAB：** 为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

Step3: 初始化零值

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

Step4: 设置对象头

初始化零值完成之后，**虚拟机要对对象进行必要的设置**，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。**这些信息存放在对象头中**。另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

Step5: 执行 init 方法

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚刚开始，`<init>` 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 `<init>` 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

HotSpot 为什么要分为新生代和老年代？

- 一般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。
- **比如在新生代中，每次收集都会有大量对象死去，所以可以选择“标记-复制”算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。**

如何判断对象是否死亡

- 引用计数法
 - 在对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器就减1；任何时刻计数器为0的对象就是不可能再被使用的。
 - 缺点：在两个对象出现循环引用的情况下，此时计数器永远不为0，导致无法对它进行回收
- 可达性算法
 - 这个算法的基本思想是通过一系列称为“**GC Roots**”的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的

垃圾收集有哪些算法，各自的特点？

- 标记-清除算法：该算法分为“标记”和“清除”阶段：首先标记出所有不需要回收的对象，在标记完成后统一回收掉所有没有被标记的对象。它是最基础的收集算法，后续的算法都是对其不足进行改进得到。
 - 缺点：效率问题和会产生内存碎片
- 复制算法：为了解决效率问题，“标记-复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。
- 标记-整理算法：根据老年代的特点提出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存，避免内存碎片的问题
- 分代收集算法

minor gc和full gc触发机制和区别

总结：

针对 HotSpot VM 的实现，它里面的 GC 其实准确分类只有两大种：

部分收集 (Partial GC)：

- - 新生代收集 (Minor GC / Young GC)：只对新生代进行垃圾收集；
 - 老年代收集 (Major GC / Old GC)：只对老年代进行垃圾收集。需要注意的是 Major GC 在有的语境中也用于指代整堆收集；
 - 混合收集 (Mixed GC)：对整个新生代和部分老年代进行垃圾收集。

整堆收集 (Full GC)：收集整个 Java 堆和方法区。

MinorGC触发条件

1. Eden区域满
2. 新创建的对象大小 > Eden所剩空间

• FullGC触发条件

老年代空间不足

- 存在情况
 1. 每次晋升到老年代的对象平均大小 > 老年代剩余空间
 2. MinorGC后存活的对象超过了老年代剩余空间

引用的种类

强引用 (Strong Reference)

类似 "Object obj = new Object()" 这类的引用，就是强引用，只要强引用存在，垃圾收集器永远不会回收被引用的对象。但是，如果我们错误地保持了强引用，比如：赋值给了 static 变量，那么对象在很长一段时间内不会被回收，会产生内存泄漏。

软引用 (Soft Reference)

软引用是一种相对强引用弱化一些的引用，可以让对象豁免一些垃圾收集，只有当 JVM 认为内存不足时，才会去试图回收软引用指向的对象。JVM 会确保在抛出 OutOfMemoryError 之前，清理软引用指向的对象。软引用通常用来实现内存敏感的缓存，如果还有空闲内存，就可以暂时保留缓存，当内存不足时清理掉，这样就保证了使用缓存的同时，不会耗尽内存。

弱引用 (Weak Reference)

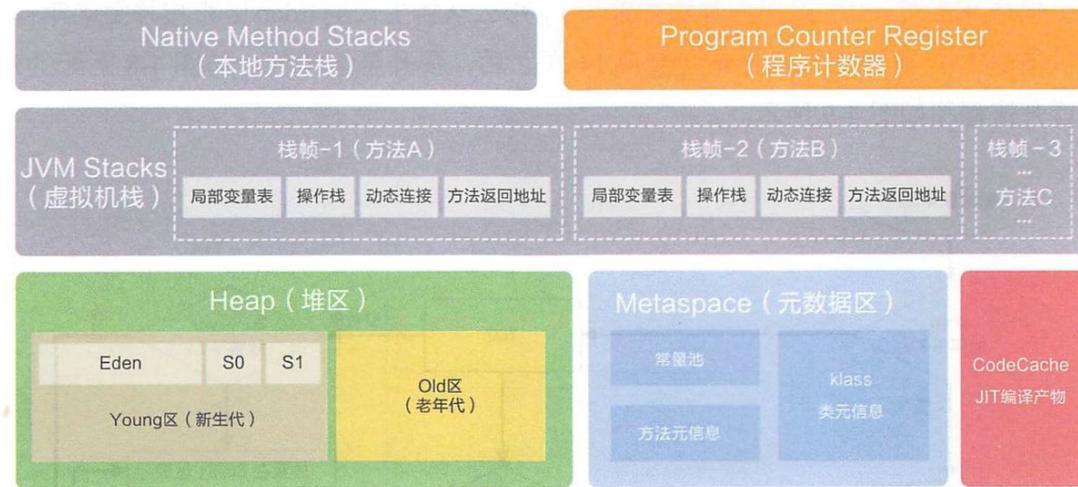
弱引用的强度比软引用更弱一些。当 JVM 进行垃圾回收时，无论内存是否充足，都会回收只被弱引用关联的对象。

虚引用 (Phantom Reference)

虚引用也称幽灵引用或者幻影引用，它是最弱的一种引用关系。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。它仅仅是提供了一种确保对象被 finalize 以后，做某些事情的机制，比如，通常用来看做所谓的 Post-Mortem 清理机制。

琐碎知识点

- 如何判断一个常量是废弃常量：当常量池中的"abc"没有任何引用引用这个常量时，这个常量就会被回收
- 如何判断一个类是无用的类：该类的所有实例被回收，ClassLoader对象被回收，class对象被回收，即没有任何对象指向这个class对象，class对象无法通过反射得到该类的方法。



并发&多线程面试题Anki

程序计数器，虚拟机栈和本地方法栈为什么是线程私有的？

- 程序计数器主要有下面两个作用：
 - 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
 - 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。
 - 所以，程序计数器私有主要是为了线程切换后能恢复到正确的执行位置。
- 虚拟机栈：
 - 每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。
 - 为了保证线程中的局部变量不被别的线程访问到，虚拟机栈和本地方法栈是线程私有的

为什么要使用多线程以及多线程带来的问题

- 在单核时代多线程主要是为了提高 CPU 和 IO 设备的综合利用率。多核时代多线程主要是为了提高 CPU 利用率。
- **从计算机底层来说：**线程可以比作是轻量级的进程，是程序执行的最小单位，线程间的切换和调度的成本远远小于进程。另外，多核 CPU 时代意味着多个线程可以同时运行，这减少了线程上下文切换的开销。
- **从当代互联网发展趋势来说：**现在的系统动不动就要求百万级甚至千万级的并发量，而多线程并发编程正是开发高并发系统的基础，利用好多线程机制可以大大提高系统整体的并发能力以及性能。
- **使用多线程可能带来什么问题：**并发编程的目的就是为了能提高程序的执行效率提高程序运行速度，但是并发编程并不总是能提高程序运行速度的，而且并发编程可能会遇到很多问题，比如：**内存泄漏、死锁、线程不安全**等等

###

说说线程的生命周期和状态

Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态：

状态名称	说 明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

- 线程创建之后它将处于 **NEW（新建）** 状态，调用 `start()` 方法后开始运行，线程这时候处于 **READY（可运行）** 状态。可运行状态的线程获得了 CPU 时间片 (timeslice) 后就处于 **RUNNING（运行）** 状态。

- 当线程执行 `wait()` 方法之后，线程进入 **WAITING (等待)** 状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 **TIME_WAITING(超时等待)** 状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep (long millis)` 方法或 `wait (long millis)` 方法可以将 Java 线程置于 TIMED WAITING 状态。当超时时间到达后 Java 线程将会返回到 RUNNABLE 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 **BLOCKED (阻塞)** 状态。线程在执行 `Runnable` 的 `run()` 方法之后将会进入到 **TERMINATED (终止)** 状态。

什么是上下文切换

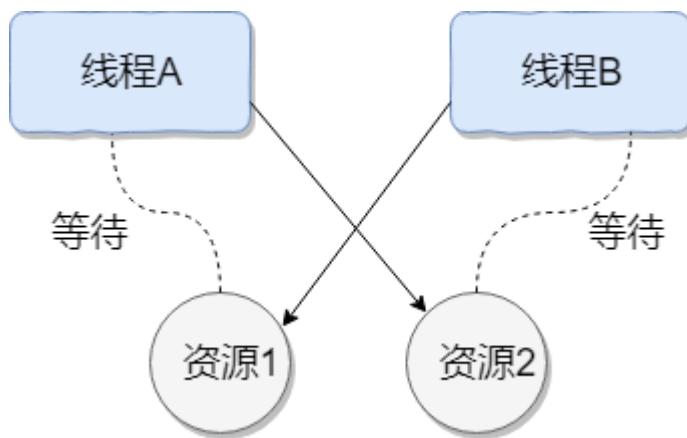
多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。

概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，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。**

什么是线程死锁？如何避免死锁？

- 线程死锁描述的是这样一种情况：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



- 如何避免死锁
 - 破坏互斥条件**：这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。
 - 破坏请求与保持条件**：一次性申请所有的资源。
 - 破坏不剥夺条件**：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
 - 破坏循环等待条件**：靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

说说 sleep() 方法和 wait() 方法区别和共同点

- 两者最主要的区别在于： `sleep()` 方法没有释放锁，而 `wait()` 方法释放了锁。
- 两者都可以暂停线程的执行。
- `wait()` 通常被用于线程间交互/通信，`sleep()` 通常被用于暂停执行。
- `wait()` 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 `notify()` 或者 `notifyAll()` 方法。`sleep()` 方法执行完成后，线程会自动苏醒。或者可以使用 `wait(long timeout)` 超时后线程会自动苏醒。

为什么我们调用 start() 方法时会执行 run() 方法，为什么我们不能直接调用 run() 方法？

new 一个 Thread，线程进入了新建状态。调用 `start()` 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。`start()` 会执行线程的相应准备工作，然后自动执行 `run()` 方法的内容，这是真正的多线程工作。但是，直接执行 `run()` 方法，会把 `run()` 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

总结：调用 `start()` 方法方可启动线程并使线程进入就绪状态，直接执行 `run()` 方法的话不会以多线程的方式执行。

synchronized 关键字的使用方式

- 修饰实例方法：**作用于当前对象实例加锁，进入同步代码前要获得 **当前对象实例的锁**
- 修饰静态方法：**也就是给当前类加锁，会作用于类的所有对象实例，进入同步代码前要获得 **当前 class 的锁**。因为静态成员不属于任何一个实例对象，是类成员（`static` 表明这是该类的一个静态资源，不管 `new` 了多少个对象，只有一份）。所以，如果一个线程 A 调用一个实例对象的非静态 `synchronized` 方法，而线程 B 需要调用这个实例对象所属类的静态 `synchronized` 方法，是允许的，不会发生互斥现象，**因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁**。
- 修饰代码块：**指定加锁对象，对给定对象/类加锁。`synchronized(this|object)` 表示进入同步代码库前要获得**给定对象的锁**。`synchronized(类.class)` 表示进入同步代码前要获得 **当前 class 的锁**
- 总结：**
 - `synchronized` 关键字加到 `static` 静态方法和 `synchronized(class)` 代码块上都是是给 Class 类上锁。
 - `synchronized` 关键字加到实例方法上是给对象实例上锁。
 - 尽量不要使用 `synchronized(string a)` 因为 JVM 中，字符串常量池具有缓存功能！

谈谈 `synchronized` 和 `ReentrantLock` 的区别以及 `ReentrantLock` 中的 `Condition`

两者都是可重入锁

“可重入锁”指的是自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增 1，所以要等到锁的计数器下降为 0 时才能释放锁。

`synchronized` 依赖于 JVM 而 `ReentrantLock` 依赖于 API

`synchronized` 是依赖于 JVM 实现的，前面我们也讲到了虚拟机团队在 JDK1.6 为 `synchronized` 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。

`ReentrantLock` 是 JDK 层面实现的（也就是 API 层面，需要 `lock()` 和 `unlock()` 方法配合 `try/finally` 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

`ReentrantLock` 比 `synchronized` 增加了一些高级功能

相比 `synchronized`，`ReentrantLock` 增加了一些高级功能。主要来说主要有三点：

- **等待可中断**：`ReentrantLock` 提供了一种能够中断等待锁的线程的机制，通过 `lockInterruptibly()` 来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。
- **可实现公平锁**：`ReentrantLock` 可以指定是公平锁还是非公平锁。而 `synchronized` 只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。`ReentrantLock` 默认情况是非公平的，可以通过 `ReentrantLock` 类的 `ReentrantLock(boolean fair)` 构造方法来制定是否是公平的。
- **可实现选择性通知（锁可以绑定多个条件）**：`synchronized` 关键字与 `wait()` 和 `notify()` / `notifyAll()` 方法相结合可以实现等待/通知机制。`ReentrantLock` 类当然也可以实现，但是需要借助于 `Condition` 接口与 `newCondition()` 方法。

关于 `Condition`

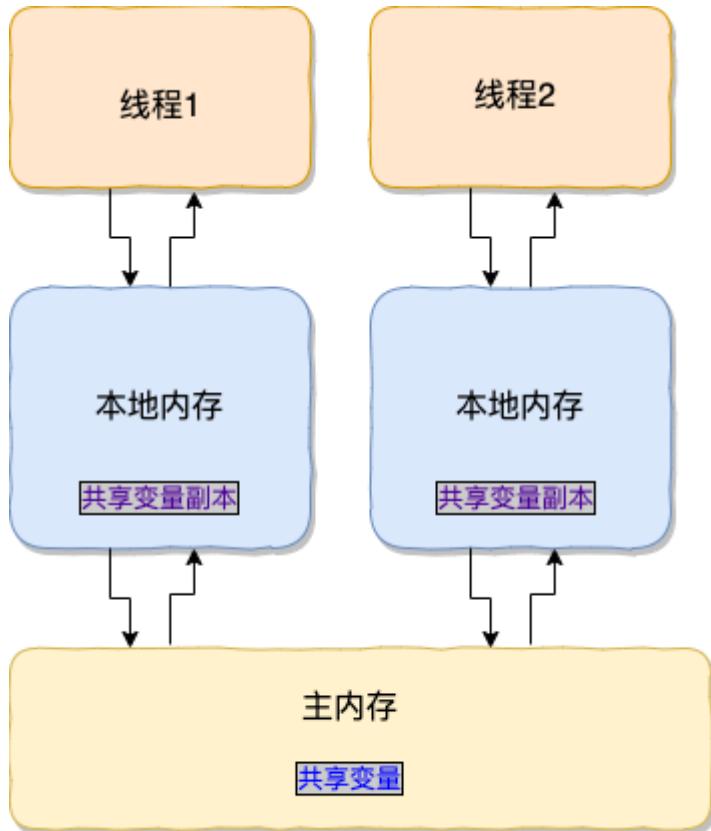
- `Condition` 是 JDK1.5 之后才有的，它具有很好的灵活性，比如可以实现多路通知功能也就是在一个 `Lock` 对象中可以创建多个 `Condition` 实例（即对象监视器），
- **线程对象可以注册在指定的 `Condition` 中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。在使用 `notify()` / `notifyAll()` 方法进行通知时，被通知的线程是由 JVM 选择的，用 `ReentrantLock` 类结合 `Condition` 实例可以实现“选择性通知”，这个功能非常重要，而且是 `Condition` 接口默认提供的。**
- 而 `synchronized` 关键字就相当于整个 `Lock` 对象中只有一个 `Condition` 实例，所有的线程都注册在它一个身上。如果执行 `notifyAll()` 方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而 `Condition` 实例的 `signalAll()` 方法只会唤醒注册在该 `Condition` 实例中的所有等待线程。

讲一下 `volatile` 关键字

- `volatile` 关键字是线程同步的轻量级实现，所以 `volatile` 性能肯定比 `synchronized` 关键字要好。但是 `volatile` 关键字只能用于变量而 `synchronized` 关键字可以修饰方法以及代码块。
- `volatile` 关键字能保证数据的可见性，但不能保证数据的原子性。`synchronized` 关键字两者都能保证。
- `volatile` 关键字主要用于解决变量在多个线程之间的可见性，而 `synchronized` 关键字解决的是多个线程之间访问资源的同步性。

讲一下 JMM(Java 内存模型)

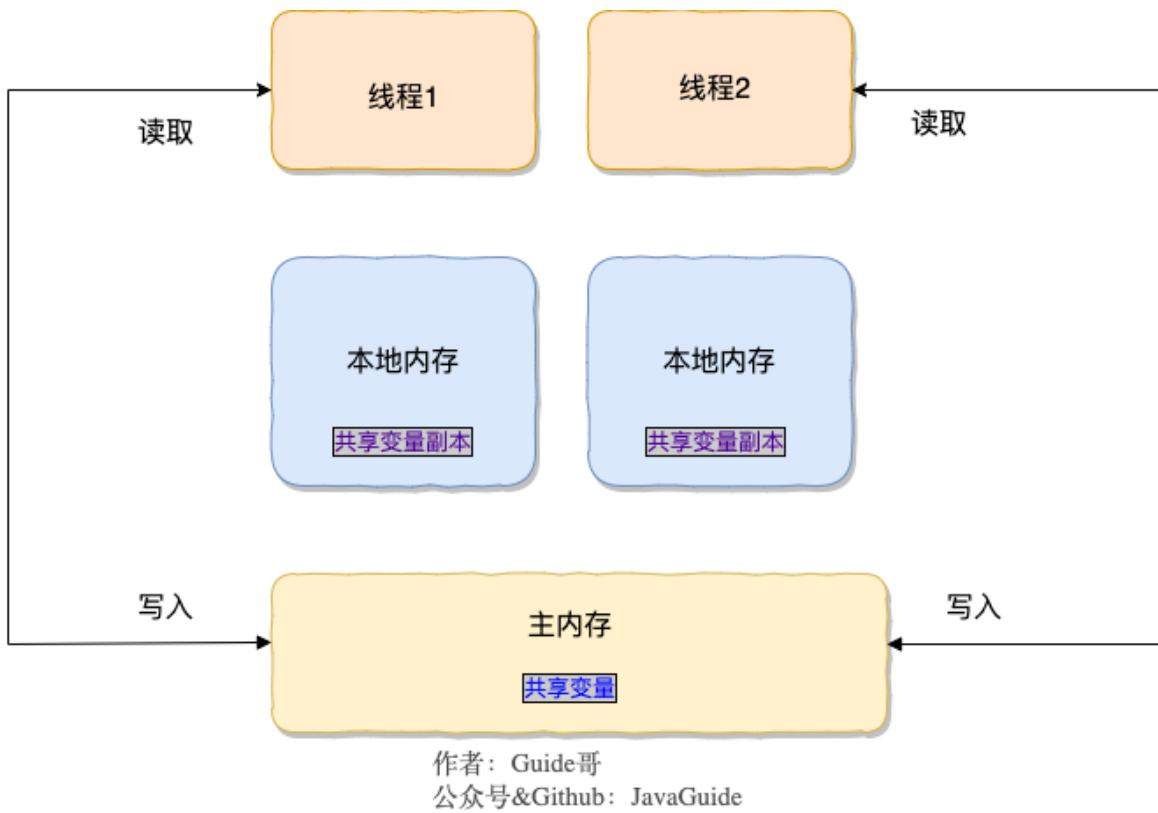
在 JDK1.2 之前，Java 的内存模型实现总是从**主存**（即共享内存）读取变量，是不需要进行特别的注意的。而在当前的 Java 内存模型下，线程可以把变量保存**本地内存**（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成**数据的不一致**。



作者: Guide哥
公众号&Github: JavaGuide

要解决这个问题，就需要把变量声明为 `volatile`，这就指示 JVM，这个变量是共享且不稳定的，每次使用它都到主存中进行读取。

所以，`volatile` 关键字除了防止 JVM 的指令重排，还有一个重要的作用就是保证变量的可见性。



并发编程的三个重要特性

- 原子性**：一个的操作或者多次操作，要么所有的操作全部都得到执行并且不会收到任何因素的干扰而中断，要么所有的操作都执行，要么都不执行。`synchronized` 可以保证代码片段的原子性。
- 可见性**：当一个变量对共享变量进行了修改，那么另外的线程都是立即可以看到修改后的最新值。`volatile` 关键字可以保证共享变量的可见性。
- 有序性**：代码在执行的过程中的先后顺序，Java 在编译器以及运行期间的优化，代码的执行顺序未必就是编写代码时候的顺序。`volatile` 关键字可以禁止指令进行重排序优化。

ThreadLocal

- 通常情况下，我们创建的变量是可以被任何一个线程访问并修改的。如果想实现每一个线程都有自己的专属本地变量该如何解决呢？JDK 中提供的 `ThreadLocal` 类正是为了解决这样的问题。`ThreadLocal` 类主要解决的就是让每个线程绑定自己的值，可以将 `ThreadLocal` 类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。
- 如果你创建了一个 `ThreadLocal` 变量，那么访问这个变量的每个线程都会有这个变量的本地副本，这也是 `ThreadLocal` 变量名的由来。他们可以使用 `get()` 和 `set()` 方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。

为什么要用线程池

- 降低资源消耗**。通过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**。当任务到达时，任务可以不需要等到线程创建就能立即执行。
- 提高线程的可管理性**。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

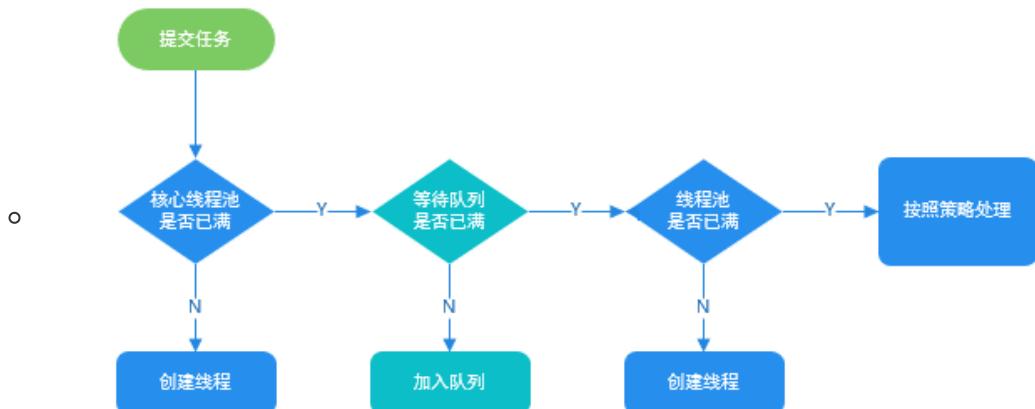
如何创建线程池以及线程池的构造方法参数的含义、饱和策略

- 通过构造方法实现或者通过 Executor 框架的工具类 Executors 来实现，但是不建议用Executor 框架的工具类 Executors 来实现
- 线程池 ThreadPoolExecutor 构造器参数的含义：

参数名	作用
corePoolSize	核心线程池大小
maximumPoolSize	最大线程池大小
keepAliveTime	线程池中超过corePoolSize数目的空闲线程最大存活时间；可以allowCoreThreadTimeOut(true)使得核心线程也会在达到keepAliveTime也将关闭
TimeUnit	keepAliveTime时间单位
workQueue	阻塞任务队列
threadFactory	新建线程工厂
RejectedExecutionHandler	当提交任务数超过maximumPoolSize+workQueue之和时，任务会交给RejectedExecutionHandler来处理

- 线程池工作流程

- 1. 当线程池小于corePoolSize时，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。注意：线程不是在ThreadPoolExecutor初始化时创建的，而是在发生外部请求调用的时候才会创建。
- 2. 当线程池达到corePoolSize时，新提交任务将被放入workQueue中，等待线程池中任务调度执行。
- 3. 当workQueue已满，且maximumPoolSize>corePoolSize时，新提交任务会创建新线程执行任务。
- 4. 当提交任务数超过maximumPoolSize时，新提交任务由RejectedExecutionHandler处理
- 5. 当线程池中超过corePoolSize线程，空闲时间达到keepAliveTime时，关闭空闲线程
- 6. 当设置allowCoreThreadTimeOut(true)时，线程池中corePoolSize线程空闲时间达到keepAliveTime也将关闭



• `ThreadPoolExecutor` 饱和策略

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任时，

`ThreadPoolExecutor` 定义一些策略：

- `ThreadPoolExecutor.AbortPolicy`：抛出 `RejectedExecutionException` 来拒绝新任务的处理。
- `ThreadPoolExecutor.CallerRunsPolicy`：调用执行自己的线程运行任务，也就是直接在调用 `execute` 方法的线程中运行(`run`)被拒绝的任务，如果执行程序已关闭，则会丢弃该任务。因此这种策略会降低对于新任务提交速度，影响程序的整体性能。如果您的应用程序可以承受此延迟并且你要求任何一个任务请求都要被执行的话，你可以选择这个策略。
- `ThreadPoolExecutor.DiscardPolicy`：不处理新任务，直接丢弃掉。
- `ThreadPoolExecutor.DiscardOldestPolicy`：此策略将丢弃最早的未处理的任务请求。

Atomic 原子类以及底层通过什么实现，如何解决ABA问题

- Atomic 翻译成中文是原子的意思。在化学上，我们知道原子是构成一般物质的最小单位，在化学反应中是不可分割的。**在我们这里 Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。底层通过CAS+volatile的原理来实现所以，所谓原子类说简单点就是具有原子/原子操作特征的类。**
- 可以将JUC包中的原子类分为4类：基本类型，数组类型，引用类型，对象的属性修改类型
- CAS ABA 问题以及如何解决：
 - 什么是CAS：当更新一个值从E->V时，更新的时候需要读取E最新的值N，如果发生了变化，也就是当E!=N，就不会更新成功，重新尝试，否则更新值成功，变为V。
 - 描述：第一个线程取到了变量x的值A，然后巴拉巴拉干别的事，总之就是只拿到了变量x的值A。这段时间内第二个线程也取到了变量x的值A，然后把变量x的值改为B，然后巴拉巴拉干别的事，最后又把变量x的值变为A（相当于还原了）。在这之后第一个线程终于进行了变量x的操作，但是此时变量x的值还是A，所以 `compareAndSet` 操作是成功。
 - 例子描述(可能不太合适，但好理解)：年初，现金为零，然后通过正常劳动赚了三百万，之后正常消费了（比如买房子）三百万。年末，虽然现金零收入（可能变成其他形式了），但是赚了钱是事实，还是得交税的！
 - 解决办法：加一个类似于版本号的东西，或者时间戳之类的。记录更新的次数即可，比较的时候不光比较value也要比较版本号。

等待 (wait) 和通知 (notify)

- 这两个方法是Object类中的，任何对象都可以调用这两个方法。
- `wait()`方法只能在`synchronized`方法或`synchronized`块中使用（原因：`wait`方法会释放锁，只有在syn中才有锁）
- `notifyAll`会让所有处于等待池的线程全部进入锁池去竞争获取锁的机会
- `notify`只会随机选取一个处于等待池中的线程进入锁池去竞争获取锁的机会

JVM对锁的优化（这个还有待修改参考下这个<https://www.cnblogs.com/wuqinglong/p/9945618.html>）

这里的锁优化主要是指 JVM 对 synchronized 的优化。

偏向锁

偏向锁的思想是偏向于让第一个获取锁对象的线程，这个线程在之后获取该锁就不再需要进行同步操作，甚至连 CAS 操作也不再需要。

当锁对象第一次被线程获得的时候，进入偏向状态，标记为 1 01。同时使用 CAS 操作将线程 ID 记录到 Mark Word 中，如果 CAS 操作成功，这个线程以后每次进入这个锁相关的同步块就不再需要再进行任何同步操作。

当有另外一个线程去尝试获取这个锁对象时，偏向状态就宣告结束，此时撤销偏向（Revoke Bias）后恢复到未锁定状态或者轻量级锁状态。如果锁竞争比较大的情况就不要使用了。

轻量级锁

如果偏向锁失败，虚拟机不会立即挂起线程，还会使用一种轻量级锁的优化手段，轻量级锁是相对于传统的重量级锁而言，它使用 CAS 操作来避免重量级锁使用互斥量的开销。对于绝大部分的锁，在整个同步周期内都是不存在竞争的，因此也就不需要都使用互斥量进行同步，可以先采用 CAS 操作进行同步，如果 CAS 失败了再改用互斥量进行同步。

如果 CAS 操作失败了，虚拟机首先会检查对象的 Mark Word 是否指向当前线程的虚拟机栈，如果是的话说明当前线程已经拥有了这个锁对象，那就可以直接进入同步块继续执行，否则说明这个锁对象已经被其他线程线程抢占了。如果有两条以上的线程争用同一个锁，那轻量级锁就不再有效，要膨胀为重量级锁。

自旋锁

自旋锁的思想是让一个线程在请求一个共享数据的锁时执行忙循环（自旋）一段时间，如果在这段时间内能获得锁，就可以避免进入阻塞状态。

自旋锁虽然能避免进入阻塞状态从而减少开销，但是它需要进行忙循环操作占用 CPU 时间，它只适用于共享数据的锁定状态很短的场景。

锁消除

锁消除是指对于被检测出不可能存在竞争的共享数据的锁进行消除。

锁消除主要是通过逃逸分析来支持，如果堆上的共享数据不可能逃逸出去被其它线程访问到，那么就可以把它们当成私有数据对待，也就可以将它们的锁进行消除。

AQS

- 目前不太理解，后面有精力再好好学习这个

琐碎知识

- **synchronized**: `synchronized` 关键字解决的是多个线程之间访问资源的同步性，
`synchronized` 关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。
-