一、知识点

1、泛型

泛型定义及常用特点

泛型是Java SE 1.5之后的特性，《Java 核心技术》中对泛型的定义是：
`“泛型”意味着编写的代码可以被不同类型的对象所重用。`

“泛型”，顾名思义，“泛指的类型”。我们提供了泛指的概念，但具体执行的时候却可以有具体的规则来约束，比如我们用的非常多的ArrayList就是个泛型类，ArrayList作为集合可以存放各种元素，如Integer，String，自定义的各种类型等，但在我们使用的时候通过具体的规则来约束，如我们可以约束集合中只存放Integer类型的元素，如：
`List<Integer>iniData=newArrayList<>()`

使用泛型的好处?

以集合来举例，使用泛型的好处是我们不必因为添加元素类型的不同而定义不同类型的集合，如整型集合类，浮点型集合类，字符串集合类，我们可以定义一个集合来存放整型、浮点型，字符串型数据，而这并不是最重要的，因为我们只要把底层存储设置了Object即可，添加的数据全部都可向上转型为Object。更重要的是我们可以通过规则按照自己的想法控制存储的数据类型。

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。因为他们接受一个或多

个参数，这些类被称为参数化的类或参数化的类型。

```
public class Box<T> {
  private T t;
  public void add(T t){
    this.t = t ;
  }

  public T get(){
    return t ;
  }
}
```

类型通配符?

类型通配符一般是使用？代替具体的类型参数。例如 `List<?>` 在逻辑上是List,List等所有 `List<具体类型实参>`的父类。

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。如在代码中定义的 List和 List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。类型擦除的基本过程也比较简单，首先是找到用来替换类型参数的具体类。这个具体类一般是 Object。如果指定了类型参数的上界的话，则使用这个上界。把代码中的类型参数都替换成具体的类。

2、异常处理

2.1、Error与Exception区别

error 表示恢复不是不可能但很困难的情况下的一种严重问题。比如说内存溢出。不可能指望程序能处理这样的情况。

exception 表示一种设计或实现问题。也就是说，它表示如果程序运行正常，从不会发生的情况。

- Error和Exception都是java错误处理机制的一部分，都继承了Throwable类。
- Exception表示的异常，异常可以通过程序来捕捉，或者优化程序来避免。
- Error表示的是系统错误，不能通过程序来进行错误处理。

2.2、Java中的Excption与Error包结构

Java可抛出(Throwable)的结构分为三种类型：

- 被检查的异常(CheckedException)
- 运行时异常(RuntimeException)
- 错误(Error)

（1）运行时异常

定义:
RuntimeException及其子类都被称为运行时异常。
特点:
Java编译器不会检查它。也就是说，当程序中可能出现这类异常时，倘若既"没有通过throws声明抛出它"，也"没有用try-catch语句捕获它"，还是会编译通过。

例如：
除数为零时产生的ArithmeticException异常，
数组越界时产生的IndexOutOfBoundsException异常，
fail-fast机制产生的ConcurrentModificationException异常等，都属于运行时异常。

常见的五种运行时异常：

- ClassCastException（类转换异常）
- IndexOutOfBoundsException（数组越界）
- NullPointerException（空指针异常）
- ArrayStoreException（数据存储异常，操作数组是类型不一致）
- BufferOverflowException

补充：fail-fast 、 Fail-safe

fail-fast
“快速失败”也就是fail-fast，它是Java集合的一种错误检测机制。
java.util包下面的所有的集合类都是快速失败的。

当多个线程对集合进行结构上的改变的操作时，有可能会产生fail-fast机制。记住是有可能，而不是一定。

例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出ConcurrentModificationException 异常，从而产生fail-fast机制，这个错叫并发修改异常。

Fail-safe
Fail-safe，java.util.concurrent包下面的所有的类都是安全失败的，在遍历过程中，如果已经遍历的数组上的内容变化了，迭代器不会抛出ConcurrentModificationException异常。如果未遍历的数组上的内容发生了变化，则有可能反映到迭代过程中。这就是ConcurrentHashMap迭代器弱一致的表现。ConcurrentHashMap的

弱一致性主要是为了提升效率，是一致性与效率之间的一种权衡。要成为强一致性，就得到处使用锁，甚至是全局锁，这就与Hashtable和同步的HashMap一样了。

（2）被检查异常

定义:
Exception类本身，以及Exception的子类中除了"运行时异常"之外的其它子类都属于被检查异常。
特点：
Java编译器会检查它。此类异常，要么通过throws进行声明抛出，要么通过try-catch进行捕获处理，否则不能通过编译。

例如:
CloneNotSupportedException就属于被检查异常。当通过clone()接口去克隆一个对象，而该对象对应的类没有实现Cloneable接口，就会抛出CloneNotSupportedException异常。被检查异常通常都是可以恢复的。

如：

- IOException
- FileNotFoundException
- SQLException

被检查的异常适用于那些不是因程序引起的错误情况，比如：读取文件时文件不存在引发的FileNotFoundExcep

（3）错误

定义：
Error类及其子类。
特点：
和运行时异常一样，编译器也不会对错误进行检查。当资源不足、约束失败、或是其它程序无法继续运行的条件发生时，就产生错误。程序本身无法修复这些错误的。

例如：
VirtualMachineError就属于错误。出现这种错误会导致程序终止运行。
OutOfMemoryError
ThreadDeath。
Java虚拟机规范规定JVM的内存分为了好几块，比如堆，栈，程序计数器，方法区等

2.3、Java中异常处理机制有几种?

异常捕捉：try...catch...finally，
异常抛出：throws

```
package com.cupdata.exception.exceptiondemo;

import
com.cupdata.exception.myexception.GenderException;

public class ExceptionDemo {

    public static void main(String[] args) {
        int a=0,b=2,c=4;
        try {
            System.out.println("程序正常执行结开始");
            System.out.println("b/a = " + (b/a));
            if (c == 4){
                throw new GenderException("throw a
new GenderException....");
            }
            System.out.println("程序正常执行结束");
        }catch (Exception e){
            System.out.println("发生异常");
            System.out.println("异常信息为: " +
e.getMessage());
            return;
        }finally {
```

```
21                 System.out.println("finally最终处理");
22             }
23         }
24 }
25
```

（1）thow与thorws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。

（2）try catch finally内的return处理顺序

- 不管有没有出现异常，finally块中代码都会执行；
- 当try和catch中有return时，finally仍然会执行；
- finally是**在return后面的表达式运算后执行的**（此时并没有返回运算后的值，而是先把要返回的值保存起来，管finally中的代码怎么样，返回的值都不会改变，任然是之前保存的值），所以函数返回值是在finally执行前确定的；

- finally中最好不要包含return，否则程序会提前退出，返回值不是try或catch中保存的返回值。

2.4、如何自定义一个异常

继承一个异常类，通常是RumtimeException或者Exception。

1、自定义异常类，继承Ecxeption类

```
package com.cupdata.exception.myexception;

public class GenderException extends Exception{
    public GenderException(String msg){
        super(msg);
    }
}

```

2、在需要抛出异常的地方使用自定义的异常类

```
package com.cupdata.exception.myexception;

public class People {
    private String name;
    private String sex;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

```

```
    public String getSex() {
        return sex;
    }

    public void setSex(String sex) throws Exception{

        if("男".equals(sex) || "女".equals(sex))
        {
            this.sex=sex;
        }
        else {
            throw new GenderException("性别必须是男或者女");//使用自定义异常类
        }
    }
}
```

3、可以在添加异常声明或者使用try...catch进行异常捕获

```
package com.cupdata.exception.myexception;

public class MyExceptionTest {
    public static void main(String[] args) throws Exception {
        People people = new People();
        people.setSex("male");
    }
}
```

或者

```java
package com.cupdata.exception.myexception;

public class MyExceptionTest {
    public static void main(String[] args) {
        People people = new People();
        try {
            people.setSex("male");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

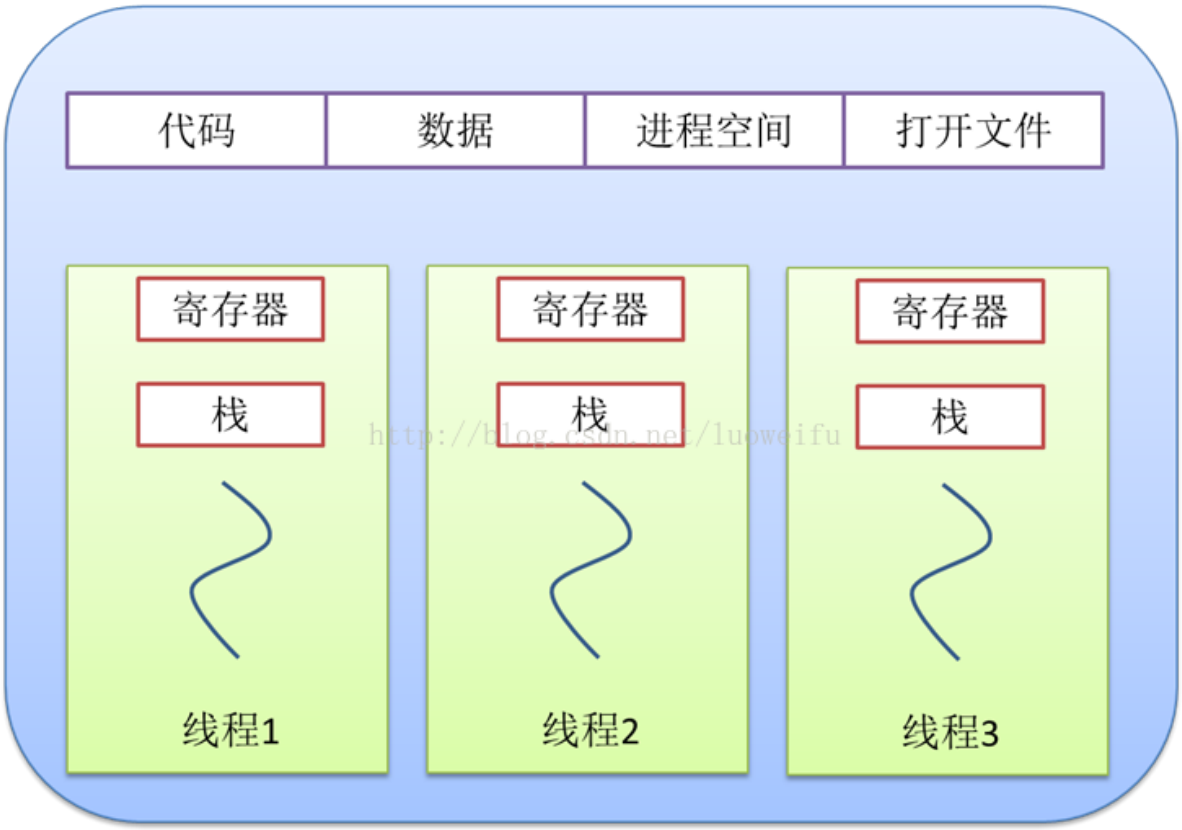
3、多线程

3.1、进程与线程

在学习Java多线程之前，先简单复习一下进程与线程的知识。

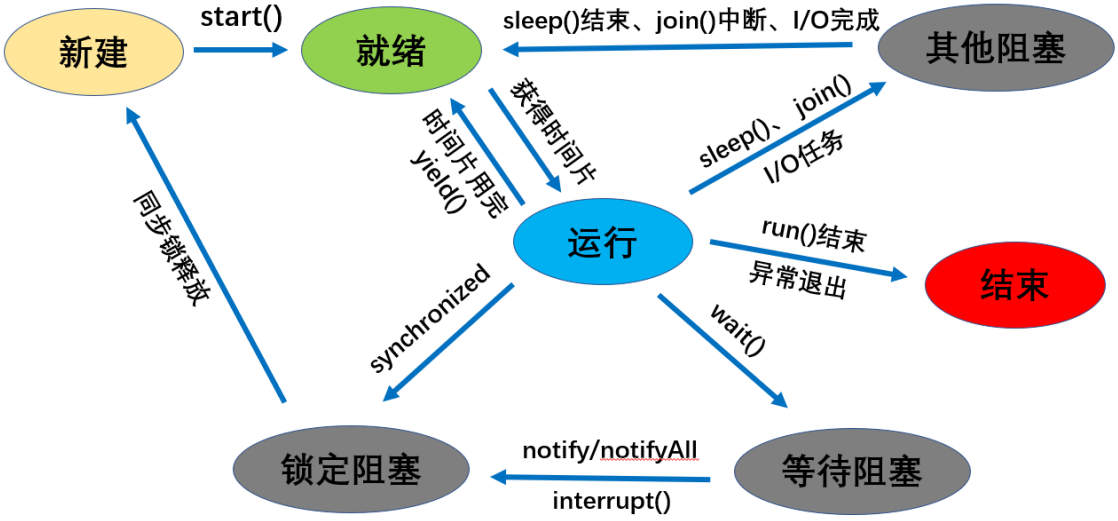
进程：进程是系统进行资源分配和调度的基本单位，可以将进程理解为一个正在执行的程序，比如一款游戏。

线程：线程是程序执行的最小单位，一个进程可由一个或多个线程组成，在一款运行的游戏中通常会有界面更新线程、游戏逻辑线程等，线程切换的开销远小于进程切换的开销。



在图1中，蓝色框表示进程，黄色框表示线程。进程拥有代码、数据等资源，这些资源是共享的，3个线程都可以访问，同时每个线程又拥有私有的栈空间。

3.2、Java线程状态图



Java线程状态图

线程的五种状态：

1）新建状态（New）：线程对象实例化后就进入了新建状态。

2）就绪状态（Runnable）：线程对象实例化后，其他线程调用了该对象的start()方法，虚拟机便会启

动该线程，处于就绪状态的线程随时可能被调度执行。

3）运行状态（Running）：线程获得了时间片，开始执行。只能从就绪状态进入运行状态。

4）阻塞状态（Blocked）：线程因为某个原因暂停执行，并让出CPU的使用权后便进入了阻塞状态。

等待阻塞：调用运行线程的wait()方法，虚拟机会把该线程放入等待池。

同步阻塞：运行线程获取对象的同步锁时，该锁已被其他线程获得，虚拟机会把该线程放入锁定池。

其他线程：调用运行线程的sleep()方法或join()方法，或线程发出I/O请求时，进入阻塞状态。

5）结束状态（Dead）：线程正常执行完或异常退出时，进入了结束状态。

3.3、Java线程实现

Java语言提供了两种实现线程的方式：

1）通过继承Thread类实现线程

```
public class ThreadTest {
    public static void main(String[] args){
        Thread thread = new MyThread(); //创建线程
        thread.start();    //启动线程
    }
}//继承Thread类
class MyThread extends Thread{
    @Override
    public void run() {
```

```
        int count = 7;
        while(count>0){
            System.out.println(count);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count--;
        }
    }
}
```

2）通过实现Runnable接口实现线程

```
public class ThreadTest {
    public static void main(String[] args){
        Runnable runnable = new MyThread();        //将Runnable对象传递给Thread构造器
        Thread thread = new Thread(runnable);
        thread.start();
    }
}//实现了Runnable接口
class MyThread implements Runnable{
    @Override
    public void run() {
        int count = 7;
        while(count>0){
            System.out.println(count);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count--;
        }
    }
```

```
22 }
```

两种方式都覆写了run()方法，run()方法内定义了线程的执行内容，我们只能通过线程的start()方法来启动线程，且start()方法只能调用一次，当线程进入执行状态时，虚拟机会回调线程的run()方法。直接调用线程的run()方法，并不会启动线程，只会像普通方法一样去执行。其实，Thread类本身也实现了Runnable接口。这两种方式都可以实现线程，但Java语言只支持单继承，如果扩展了Thread类就无法再扩展其他类，远没有实现接口灵活。

3.4、线程常用方法

1）Thread类

Thread()：用于构造一个新的Thread。

Thread(Runnable target)：用于构造一个新的Thread，该线程使用了指定target的run方法。

Thread(ThreadGroup group,Runnable target)：用于在指定的线程组中构造一个新的Thread，该

线程使用了指定target的run方法。

currentThread()：获得当前运行线程的对象引用。

interrupt()：将当前线程置为中断状态。

sleep(long millis)：使当前运行的线程进入睡眠状态，睡眠时间至少为指定毫秒数。

join()：等待这个线程结束，即在一个线程中调用other.join()，将等待other线程结束后才继续本线程。

yield()：当前执行的线程让出CPU的使用权，从运行状态进入就绪状态，让其他就绪线程执行。

2）Object类

wait()：让当前线程进入等待阻塞状态，直到其他线程调用了此对象的notify()或notifyAll()方法后，当

前线程才被唤醒进入就绪状态。

notify()：唤醒在此对象监控器上等待的单个线程。

notifyAll()：唤醒在此对象监控器上等待的所以线程。

注：wait()、notify()、notifyAll()都依赖于同步锁，而同步锁是对象持有的，且每个对象只有一个，所以

这些方法定义在Object类中，而不是Thread类中。

3）yield()、sleep()、wait()比较

wait()：让线程从运行状态进入等待阻塞状态，并且会释放它所持有的同步锁。

yield()：让线程从运行状态进入就绪状态，不会释放它锁持有的同步锁。

sleep()：让线程从运行状态进入阻塞状态，不会释放它锁持有的同步锁。

3.5、线程同步

先看一个多线程模拟卖票的例子，总票数7张，两个线程同时卖票：

```
public class ThreadTest{
    public static void main(String[] args){
        Runnable r = new MyThread();
        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
    }
}
class MyThread implements Runnable{
```

```
    private int tickets = 7;   //票数
    @Override
    public void run(){
        while(tickets>0){
            System.out.println("tickets:"+tickets);
            tickets--;
            try{
                Thread.sleep(100);
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }
    }
}
```

运行结果不符合我们的预期，因为两个线程使用共享变量tickets，存在着由于交叉操作而破坏数据的可能性，这种潜在的干扰被称作临界区，通过同步对临界区的访问可以避免这种干扰。在Java语言中，每个对象都有与之关联的同步锁，并且可以通过使用synchronized方法或语句来获取或释放这个锁。在多线程协作时，如果涉及到对共享对象的访问，在访问对象之前，线程必须获取到该对象的同步锁，获取到同步锁后可以阻止其他线程获得这个锁，直到持有锁的线程释放掉锁为止。

```
public class ThreadTest{
    public static void main(String[] args){
        Runnable r = new MyThread();
        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
    }
}
class MyThread implements Runnable{
    private int tickets = 7;
    @Override
    public void run(){
        while(tickets>0){
```

```
15              synchronized(this){  //获取当前对象的同步锁
16                  if(tickets>0){
17
 System.out.println("tickets:"+tickets);
18                      tickets--;
19                      try{
20                          Thread.sleep(100);
21                      }catch(InterruptedException e){
22                          e.printStackTrace();
23                      }
24                  }
25              }
26          }
27      }
28  }
```

3.6、锁

java中有两种锁，一种是重量级锁synchronized，jdk1.6经过锁优化加入了偏向锁和轻量级锁，一种是JUC并发包下的Lock锁，synchronized锁也称对象锁，每个对象都有一个对象锁。

锁的作用是保证可见性，原子性，有序性，具体概念**了解即可**：

可见性概念
可见性（Visibility）：是指一个线程对共享变量进行修改，另一个先立即得到修改后的最新值。

并发编程时，会出现可见性问题，当一个线程对共享变量进行了修改，另外的线程并没有立即看到修改后的最新值。

原子性概念
原子性（Atomicity）：在一次或多次操作中，要么所有的操作都执行并且不会受其他因素干扰而中断，要么所有的操作都不执行。

并发编程时，会出现原子性问题，当一个线程对共享变量操作到一半时，另外的线程也有可能来操作共享变量，干扰了前一个线程的操作。

有序性概念
有序性（Ordering）：是指程序中代码的执行顺序，Java在编译时和运行时会对代码进行优化，会导致程序最终的执行顺序不一定就是我们编写代码时的顺序。

程序代码在执行过程中的先后顺序，由于Java在编译期以及运行期的优化，导致了代码的执行顺序未必就是开发者编写代码时的顺序。

（1）synchronized用法：

1）synchronized方法：如果一个线程要在某个对象上调用synchronized方法，那么它必须先获取这个对象的

锁，然后执行方法体，最后释放这个对象上的锁，而与此同时，在同一个对象上调用synchronized方法的其他

线程将阻塞，直到这个对象的锁被释放为止。

```
public synchronized void show(){
    System.out.println("hello world");
}
```

2）synchronized静态方法：静态方法也可以被声明为synchronized的，每个类都有与之相关联的Class对象，

而静态同步方法获取的就是它所属类的Class对象上的锁，两个线程不能同时执行同一个类的静态同步方法，如

果静态数据是在线程之间共享的，那么对它的访问就必须利用静态同步方法来进行保护。

3）synchronized语句：静态语句可以使我们获取任何对象上的锁而不仅仅是当前对象上的锁，也能够让我们定

义比方法还要小的同步代码区，这样可以让线程持有锁的时间尽可能短，从而提高性能。

```
private final Object lock = new Object();
public void show(){
     synchronized(lock){
          System.out.println("hello world");
     }
}
```

（2）JUC（了解）

在 Java 5.0 提供了 java.util.concurrent （简称JUC ）包，在此包中增加了在并发编程中很常用的实用工具类，用于定义类似于线程的自定义子系统，包括线程池、异步 IO 和轻量级任务框架。提供可调的、灵活的线程池。还提供了设计用于多线程上下文中的 Collection 实现等。

有以下基本概念需要明确：

1. **内存可见性**

   内存可见性（Memory Visibility）是指当某个线程正在使用对象状态而另一个线程在同时修改该状态，需要确保当一个线程修改了对象状态后，其他线程能够看到发生的状态变化。
   可见性错误是指当读操作与写操作在不同的线程中执行时，我们无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。
   我们可以通过同步来保证对象被安全地发布。除此之外我们也可以使用一种更加轻量级的 volatile变量。

2. **volatile 关键字**

   Java 提供了一种稍弱的同步机制，即 volatile 变量，用来确保将变量的更新操作通知到其他线程。可以将 volatile 看做一个轻量级的锁，但是又与锁有些不同：

   对于多线程，不是一种互斥关系
   不能保证变量状态的“原子性操作”

3. **CAS算法**
   CAS (Compare-And-Swap) 是一种硬件对并发的支持，针对多处理器操作而设计的处理器中的一种特殊指令，用于管理对共享数据的并发访问。
   CAS 是一种无锁的非阻塞算法的实现。
   CAS 包含了 3 个操作数：需要读写的内存值 V、进行比较的值 A、拟写入的新值 B
   当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作。

   具体应用见“三、参考资料——2”

二、实战

1、改造分期功能

三、参考资料

1、泛型

java 泛型详解-绝对是对泛型方法讲解最详细的，没有之一 - little fat - 博客园 (cnblogs.com)

2、JUC

Java 之 JUC - 小a的软件思考 - 博客园 (cnblogs.com)

3、多线程

并发容器的注意事项 - 被罚站的树 - 博客园 (cnblogs.com)

4、其他

AobingJava/JavaFamily: 【Java面试+Java学习指南】 一份涵盖大部分Java程序员所需要掌握的核心知识。 (github.com)