计算机中，每个软件运行一次，即启动了一个进程。相当于整个泡菜的过程，是一个进程；而每个**统筹方法**工序，是一个个线程。**统筹方法**工序的工作模式，称之为**多线程**。

继承Thread类

**public** **class** **Person** **extends** **Thread** {

**@Override**

**public** **void** **run**() {

**try** {

System.out.println(getName() + " 开始取钱");

Thread.sleep(200);//sleep（）方法的作用是让进程睡眠、暂时不再继续执行，交出CPU，让CPU去执行其他的任务。

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.out.println(getName() + " 取钱完毕");

}

}

**public** **class** **Bank** {

**public** **static** **void** **main**(String[] args) {

Person thread1 = **new** Person();

thread1.setName("张三");

Person thread2 = **new** Person();

thread2.setName("李四");

thread1.start();

thread2.start();

}

}

Thread父类中有name属性，但是private的，所以可以调用setName（）方法为线程设置名字，通过getName（）就知道是哪个线程在运行。

线程类的run（）方法是系统调用start（）后自动执行的，编程时不需要调用run（）方法。但永远无法知道系统在什么时刻调用，是立即调用，还是延迟一小段时间调用，都由系统自动决定，无法干预。

实现Runnable接口

继承Thread类定义多线程程序后，缺点就比较明显了，无法再继承其它类，因为Java是单继承的，只允许继承一个类，这会导致程序的可拓展性大大降低。

所以定义多线程程序，优先采用第二种方式：实现java.lang.Runnable接口。

**public** **class** **Person** **implements** **Runnable** {

**private** String name;

**public** String **getName**() {

**return** name;

}

**public** **void** **setName**(String name) {

**this**.name = name;

}

**@Override**

**public** **void** **run**() {

**try** {

System.out.println(name + " 开始取钱");

Thread.sleep(200);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.out.println(name + " 取钱完毕");

}

}

Runnable接口中只有一个待实现的run（）方法，要自己补充属性了。

实现了Runnable接口的线程类，还需要包装在Thread类的实例中运行：

**public** **class** **Bank** {

**public** **static** **void** **main**(String[] args) {

Person person1 = **new** Person();

person1.setName("张三");

Thread thread1 = **new** Thread(person1);

Person person2 = **new** Person();

person2.setName("李四");

Thread thread2 = **new** Thread(person2);

thread1.start();

thread2.start();

}

}

Thread实例（new Thread（person1））相当于调度器，触发线程任务执行，线程类的实例（new Person（））就相当于任务。任务是不能自己启动的，需要被调度。

线程安全

车票类

车票类的主要作用是控制车票的总数。每售卖一次，票数减一。

**public** **class** **Ticket** {

**private** **int** count = 30;

**public** **void** **sell**() {

count--;

System.out.println(Thread.currentThread().getName() + "：卖出一张，还剩下 " + count + " 张票");

}

**public** **int** **getCount**() {

**return** count;

}

}

售票窗口类

售票窗口就是线程类，以多线程的方式售票。售票简单来说就是循环减一即可。当然了，票数为0的时候不能再售卖了。

这里假定打印票据需要点时间，售卖一张票休息100ms。

**public** **class** **TicketWindow** **implements** **Runnable** {

**private** Ticket ticket;

**public** **TicketWindow**(Ticket ticket) {

**this**.ticket = ticket;

}

**@Override**

**public** **void** **run**() {

**while** (ticket.getCount() > 0) {

**try** {

Thread.sleep(100);

ticket.sell();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

火车站类

火车站类有四个窗口，所以启动四个线程。

**public** **class** **TrainStation** {

**public** **static** **void** **main**(String[] args) {

Ticket ticket = **new** Ticket();

**for** (**int** i = 1; i <= 4; i++) {

TicketWindow office = **new** TicketWindow(ticket);

Thread thread = **new** Thread(office);

thread.setName("售票窗口" + i);

thread.start();

}

}

}

新知识点

这里有一个新的用法：Thread.currentThread().getName()

Thread.currentThread() 返回当前正在运行的线程的实例对象，因为一个车票实例，是运行在多个线程中的，需要知道具体是哪个窗口售卖车票，就可以使用这个方法。

再调用线程的 getName（） 方法可以取得当前线程实例的名称。这个名称可以使用setName（）方法指定一个名称，如果没有指定，系统会默认指定一个名称。

问题一：余量是错乱的，甚至可能余量相同

多个线程 操作 同一个资源 的时候，发生了冲突的现象，就叫做 **线程不安全**

在Java中，可以用synchronized 关键字来解决余量错乱的问题。

Synchronized加载方法上，紧跟着public：

**public** **class** **Ticket** {

**public** **synchronized** **void** **sell**() {

count--;

System.out.println(Thread.currentThread().getName() + "：卖出一张，还剩下 " + count + " 张票");

}

}

Synchronized也叫线程 **同步锁**，表示此方法是锁定的，同一时刻只能由一个线程执行此方法。

问题二：余量是负数

当余量为1的时候，四个线程可能同时判断ticket.getCount() > 0调节成立，所以，真正执行sell（）放入时候，继续减1，出现负数。

所以，对于sell（）方法来说，必须保持逻辑完整性，不能依赖其它类的条件判断，自己就不判断了。

**public** **class** **Ticket** {

**public** **synchronized** **void** **sell**() {

**if** (count > 0) {

count--;

}

System.out.println(Thread.currentThread().getName() + "：还剩下 " + count + " 张票");

}

}

Synchronized使用场景

从抽象概念来说，使用了synchronized的方法意味着满足了两个线程安全的特性：

1.**原子性：**方法全部执行并且执行的过程不会被任何因素打断。

2.**可见性：**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。

但是synchronized 为了实现这两个特性，也是要付出代价的：性能可能不高。因为方法加锁，同时只有一个线程竞争成功能继续执行，其它很多线程是持续等待、响应慢的。所以synchronized 不能滥用，比较适合的场景是：  
1.写操作的场景。例如用户修改个人信息、点赞、收藏、下单等。

2.尽量精确锁住最小的代码块，把最关键的写操作抽象成独立的方法加锁。不建议给大段的方法加锁。

悲观锁和乐观锁

上一节讲到，synchronized 可能由于线程等待导致性能问题。那我们尝试换一种**思路**。

Java.util.concurrent 是java系统提供的并发编程包，尝试使用java.util.concurrent.atomic.AtomicInteger 让车票余量能够安全的递减。

**import** java.util.concurrent.atomic.AtomicInteger;

**public** **class** **Ticket** {

**private** AtomicInteger count = **new** AtomicInteger(30);

**public** **void** **sell**() {

**int** newCount = 0;

**if** (count.get() > 0) {

newCount = count.decrementAndGet();

}

System.out.println(Thread.currentThread().getName() + "：还剩下 " + newCount + " 张票");

}

**public** **int** **getCount**() {

**return** count.get();

}

}

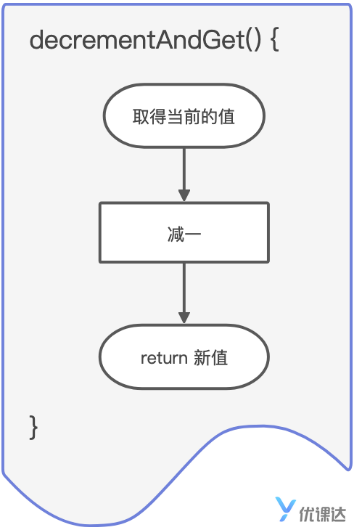
解决的问题

Sell（）不再加锁，解决了车票余量重复的问题。

AtomicInteger 虽然是一个类，但类等同于一个整数（就像Integer是int的对象）。调用new AtomicInteger（）构造函数实例化对象的时候，可以指定任意的整数值。

（new AtomicInteger（30）意思是设定实例的整数值为30）

不同的是，AtomicInteger 提供了不使用synchronized 就能保证数据操作原子性的方法。例如decrementAndGet（）方法：

decrementAndGet（）方法是三个操作的组合，多线程情况下也不会出现数值重复的错误，保证这三个操作是密不可分的、线程间没有互相干扰打断，保证了数据的准确性。这就是类名Atomic – **原子性** 的含义。

线程间都是基于**最新的**结果进行减一的运算，所以不会重复，这样是**可见性**的体现。

（count.decrementAndGet()代替了使用synchronized时，整数count--）

没有解决的问题

从输出结果可以看到，依然可能出现负数，而且Console打印的顺序也可能错误。这是因为条件判断语句、操作语句、打印信息语句组合起来，就不具备原子性了，因为sell（）不加锁，多条语句执行时就可能被其它线程打断了。

所以必须给sell（）整体加synchronized才能保证多条语句整体的原子性：

**public** **synchronized** **void** **sell**() {

}

另一个原子性的案例

春蕾中学新生入学报名的时候，会自动获得一个学号。新生注册可能是网上注册、或者校园现场新生接待处的窗口排队注册。其实相当于多线程并发情况下，要保证学号不能重复。

同样的，可以使用AtomicInteger实现。先看一下核心代码：

**import** java.util.concurrent.atomic.AtomicInteger;

**public** **class** **Register** **implements** **Runnable** {

**private** **static** AtomicInteger count = **new** AtomicInteger(0);

**private** Student student;

**public** **Register**(Student student) {

**this**.student = student;

}

**@Override**

**public** **void** **run**() {

student.setId(count.incrementAndGet());

System.out.println("姓名：" + student.getName() + "，学号：" + student.getId());

}

}

乐观锁和悲观锁

乐观锁其实就是不上锁，总是保证基于最新的数据进行更新。由于没有上锁，就**提高了性能**。不上锁的思想是乐观的，所以称之为乐观锁。AtomicInteger类的incrementAndGet（）和decrementAndGet（）方法就是典型的乐观锁实现。

相对的，synchronized关键字是把整个方法执行前就上锁，**假设**其他线程**一定会修改**数据，所以**提前防范**。上锁的思想是悲观的，所以称之为悲观锁。

（乐观锁和悲观锁是面试过程中出现概率很高的知识点哦。）

乐观锁

不适用于多条数据需要修改、以及多个操作的整体顺序要求很严格的场景，乐观锁适用于读数据比重更大的应用场景；反之，

悲观锁

适合写数据比重更大的应用场景。一般来说写数据的整体消耗时间更长些，是可以接受的。

一种思想

乐观锁/悲观锁 实际上是一种思想，不是 java领域特有的概念。在其它领域，例如数据库系统中也有 乐观锁/悲观锁 的概念。

并发容器

**CompletableFuture应用**

CompletableFuture是一个异步任务编排、调度框架，以更优雅的方式实现组合式异步编程。

（在方法调用的时候（就是所谓任务），需要等待返回取得返回值是同步，不等待而继续执行程序代码就是异步。采用异步方式，能够支持多个任务并行执行，这种机制称为并发。）

1.Register类重构

**public** **class** **Register** {

**private** **static** AtomicInteger count = **new** AtomicInteger(0);

*// 注册学号*

**public** Student **regId**(Student student) {

student.setId(count.incrementAndGet());

**return** student;

}

}

2.并行注册

**public** **class** **StudentIDTest** {

**public** **static** **void** **main**(String[] args) {

*// 构建学生集合*

List<Student> studentList = **new** ArrayList<>();

**for** (**int** i = 1; i <= 10; i++) {

Student s = **new** Student();

s.setName("学生" + i);

studentList.add(s);

}

Register reg = **new** Register();

studentList.forEach(s -> {

CompletableFuture.supplyAsync(

*// 每个学生都注册学号*

() -> reg.regId(s)

)

*// 学号注册完毕后，打印欢迎消息*

.thenAccept(student -> {

System.out.println("你好 " + student.getName() + ", 欢迎来到春蕾中学大家庭");

});

});

System.out.println("mission complate");

}

}

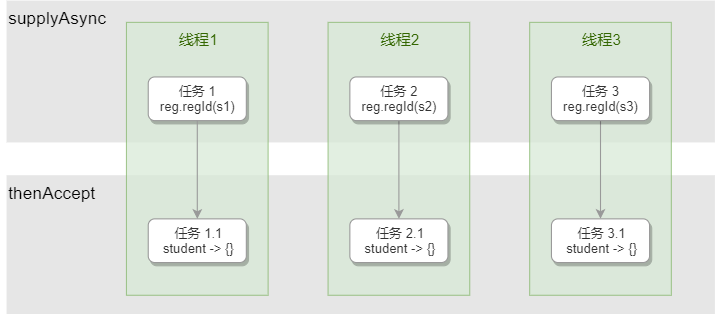
CompletableFuture.supplyAsync( )方法运行一个异步任务并且返回结果，所以regId（）方法必须有返回值。

Register虽然没有实现Runnable接口，但系统会自动优化：把作为supplyAsync（）方法参数的整个（）->reg.regId(s) 表达式语句包装在另一个对象中；这个对象也是JDK内置的，它实现了Runnable接口，在这个对象中执行表达式语句。

所以实际上，supplyAsync（）方法的作用是：在一个单独的线程中执行reg.regId(s)语句，本质上就是多线程编程。

在注册完毕后，使用thenAccept（）方法完成后继的任务步骤。ThenAccept（）方法的参数（student）就是前置任务的返回结果，系统会在前一个任务完成后，自动执行student->{ }后继任务。所以本质上，后继任务也是多线程方式执行的。ThenAccept（）方法通常用于任务链的末尾。

（一般情况下，不建议一个大任务的所有步骤都集中在supplyAsync（）里实现，分步骤更好）



**多步骤任务**

CompletableFuture.supplyAsync(() -> reg.regId(s))

.thenApply(student -> {

**return** dis.assignClasses(student);

})

.thenAccept(student -> {

System.out.println("姓名：" + student.getName() + "，学号：" + student.getId() + "，班级号：" + student.getClassId());

});

SupplyAsync（）用于**开头**，thenAccept（）用于**末尾**，各自调用一次即可。中间有多个步骤，可以调用多次thenApply（）。由于末尾也要用到student实例对象，所以位于中间的thenApply（）方法，总要return学生实例对象，否则下一个步骤就获取不到。

**拓展知识点：返回值**

SupplyAsync（）是静态方法，返回值是CompletableFuture实例对象，再调用thenApply（）或者thenAccept（）实例方法，返回的也是CompletableFuture实例对象。

所以，虽然整条语句是连写的，其实也可以定义返回值。

CompletableFuture<Void> cf = CompletableFuture.supplyAsync(() -> reg.regId(s))

.thenApply(student -> {

**return** dis.assignClasses(student);

})

.thenAccept(student -> {

System.out.println("姓名：" + student.getName() + "，学号：" + student.getId() + "，班级号：" + student.getClassId());

});

返回的仍然是CompletableFuture实例对象，所以定义变量的类型就是CompletableFuture。但可以用范型CompletableFuture<>表示其中包含的数据具体是什么类型。

因为本案例末尾调用了thenAccept（），其Lambda表达式没有return 语句，表示completableFuture实例对象不包含数据，所以泛型写为CompletableFuture<Void>。

**返回CompletableFuture类型**

如果没有调用thenAccept（）方法，以thenApply（）或者supplyAsync（）结尾的话，例如代码：

CompletableFuture.supplyAsync(() -> reg.regId(s))

.thenApply(student -> {

**return** dis.assignClasses(student);

});

因为thenApply（）的Lambda表达式返回的是Student对象，所以CompletableFuture实例对象包含的是Student数据，于是泛型写为CompletableFuture<Student>

CompletableFuture<Student> cf = CompletableFuture.supplyAsync(() -> reg.regId(s))

.thenApply(student -> {

**return** dis.assignClasses(student);

});

**拓展知识点：main（）方法的问题**

当线程任务数量多的时候，可能线程任务还没执行完毕，mian（）方法就执行完毕，导致程序运行结束退出了。

要解决这个问题，返回值就有用了。我们先把每个学生的入学任务实例对象（CompletableFuture<void>），收集起来（装入集合），然后等待所有的线程执行完毕。

List<CompletableFuture> cfs = **new** ArrayList<>();

studentList.forEach(s -> {

CompletableFuture<Void> cf = CompletableFuture.supplyAsync(() -> reg.regId(s))

.thenApply(student -> {

**return** dis.assignClasses(student);

}).thenAccept(student -> {

System.out.println("姓名：" + student.getName() + "，学号：" + student.getId() + "，班级号：" + student.getClassId());

});

cfs.add(cf);

});

**try** {

*// 等待所有的线程执行完毕*

CompletableFuture.allOf(cfs.toArray(**new** CompletableFuture[] {})).get();

} **catch** (Exception e) {

e.printStackTrace();

}

CompletableFuture.allOf（）是静态方法，作用是收集所有的任务实例对象。因为allOf（）方法只支持数组不支持集合，所以需要把集合转换成数组（cfs.toArray(new CompletableFuture[ ]{ })）。当然，也可以一开始就定义数组来收集任务实例对象，因为学生的个数可以通过studentList.size( )取得。AllOf（）方法的返回值也是CompletableFuture实例对象。

再调用类方法get（），其作用就是等待所有的任务进程（allOf（）收集的）都执行完毕，再继续执行mian（）方法后面的程序。

**知识点1：安全的布尔值包装类**

Java.util.concurrent.atomic.AtomicInteger能够以原子的方式操作整数，Java也提供了java.util.concurrent.atomic.AtomicBoolean能够以原子的方式操作布尔值。

因为布尔值只有ture/false两个值，AtomicBoolean使用起来就很简便了。

AtimocBoolean是boolean的包装类，AtomicBoolean的实例等同于一个布尔值：

\*new AtomicBoolean（true）等同于 **true**

\*new AtomicBoolean（false）等同于 **false**

取得布尔值

实例对象取得基础类型的布尔值，可以调用get（）方法：

AtomicBoolean ab = **new** AtomicBoolean(**true**);

**boolean** value = ab.get();

实例对象调用compareAndSet（）方法，就能以原子的方式修改值：

True 改为 false

CompareAndSet（true，false）判断当前值为true时，修改为false，然后返回成功或失败。这是三个步骤。

\*修改成功后，方法返回true。

\*如果当前值不是true，则不修改，返回值为false，表示操作失败。

False 改为 true

CompareAndSet（false）判断当前值为false时，修改为true，然后返回成功或失败。这是三个步骤。

\*修改成功后，方法返回true。

\*如果当前值不是false，则不修改，返回值为false，表示操作失败。

**再次强调**：compareAndSet（）方法返回值表示**修改操作**成功或失败，跟方法参数无关。

线程池

使用Runable接口开发多线程程序，更符合面向对象的习惯，但是随之而来的问题是，对象太多。

本章第五节课为了方便，只演示了4位同学注册的场景，但实际上在现实中，入学人数肯定不是4个这么少。如果有一千位同学入学，就意味着程序要额外 new Thread（register）一千次。对象除了创建需要消耗计算机CPU、内存等资源，对象还会被销毁（系统自己做的），销毁也是要消耗资源的。

那么能不能做一些优化，做到复用Thread对象，不必每次都创建新对象呢？  
答案是有的：Java提供了 **线程池**技术。

**线程池基本概念**

所谓**线程池**，顾名思义，就像一个池子，里面装满了线程，随用随取。线程可以被**复用**，一个线程，可以执行A任务，也可以执行B任务，于是线程不再频繁创建和销毁。

（new Thread（register）意味着一个线程对象只能执行一个任务，而线程池让线程与任务分离，不再紧密绑定。）

**线程池创建**

**import** org.apache.commons.lang3.concurrent.BasicThreadFactory;

**import** java.util.concurrent.\*;

**public** **class** **StudentIDTest** {

*// 线程工厂*

**private** **static** **final** ThreadFactory namedThreadFactory = **new** BasicThreadFactory.Builder()

.namingPattern("studentReg-pool-%d")

.daemon(**true**)

.build();

*// 等待队列*

**private** **static** **final** BlockingQueue<Runnable> workQueue = **new** LinkedBlockingQueue<Runnable>(1024);

*// 线程池服务*

**private** **static** **final** ThreadPoolExecutor EXECUTOR\_SERVICE = **new** ThreadPoolExecutor(

20,

200,

30,

TimeUnit.SECONDS,

workQueue,

namedThreadFactory,

**new** ThreadPoolExecutor.AbortPolicy()

);

**public** **static** **void** **main**(String[] args) {

**for** (**int** i = 1; i <= 2000; i++) {

Student s = **new** Student();

s.setName("学生" + i);

Register register = **new** Register(s);

*// 传入 Runnable 对象，运行任务*

EXECUTOR\_SERVICE.execute(register);

}

}

}

这里的BasicThreadFactory需要依赖一个库，这个库也是经常会使用的工具库：

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.10</version>

</dependency>

**创建线程池的代码，基本上属于固定写法**，有不懂的类和方法，也没关系，**不要太纠结**。

**一、创建线程工厂实例**

**new** BasicThreadFactory.Builder()

.namingPattern("studentReg-pool-%d")

.daemon(**true**)

.build();

唯一**需要注意**的是，namingPattern（）方法是定义线程名字的格式，相当于线程名称模块，需要根据具体的业务需求把“**studentReg**”改掉。例如商品发布可以命名为“offer-pool-%d”。

**二、创建线程等待队列实例**

*// 等待队列*

**private** **static** **final** BlockingQueue<Runnable> workQueue = **new** LinkedBlockingQueue<Runnable>(1024);

如果机器性能好，CPU核数多（6、8核）、内存大，队列可以大一些：new LinkedBlockingQueue<Runnable>(2048)。构造函数的参数表示能排队的任务个数。

如果机器性能不好，CPU核数少（1、2核）、内存小，队列可以小一些：new LinkedBlockingQueue<Runnable>(512)。

一般来说，new LinkedBlockingQueue<Runnable>(1024)也还可以。

**三、创建线程池实例**

*// 线程池服务*

**private** **static** **final** ThreadPoolExecutor EXECUTOR\_SERVICE = **new** ThreadPoolExecutor(

20,

200,

30,

TimeUnit.SECONDS,

workQueue,

namedThreadFactory,

**new** ThreadPoolExecutor.AbortPolicy()

);

ThreadPoolExecutor构造函数参数较多，七个参数按顺序说明如下：

参数序号 解释

1 线程池初始化核心线程数量，一般是两位数，通常不大

2 线程池最大线程数，计算机性能强就大一些,否则小一些，通常不超过200

3 线程池中的线程数超过核心线程数时，如果一段时间后还没有任务指派，就回收了。想立即回收就填0，一般30

4 第三个参数的时间单位。30 +TimeUnit.SECONDS表示30秒

5 等待队列实例，已经创建过了

6 线程工厂实例，已经创建过了

7 任务太多，超过队列的容量时，用什么样的策略处理。一般用AbortPolicy表示拒绝，让主线程自己处理

**使用线程池运行任务**

**public** **class** **StudentIDTest** {

**public** **static** **void** **main**(String[] args) {

*// 构建学生集合*

**for** (**int** i = 1; i <= 2000; i++) {

Student s = **new** Student();

s.setName("学生" + i);

Register register = **new** Register(s);

*// 传入 Runnable 对象，运行任务*

EXECUTOR\_SERVICE.execute(register);

}

}

}

线程池与并发容器

实际上，CompletableFuture内部也用到了线程池。

CompletableFuture.supplyAsync(

() -> reg.regId(s)

)

实际上是把任务放入内部的**默认线程池**里执行的。

CompletableFuture也可以指定线程池来运行任务：

CompletableFuture.supplyAsync(

() -> reg.regId(s),

EXECUTOR\_SERVICE

)

supplyAsync（）方法可以有第二个参数，传入我们构建好的线程池对象。那么任务就是用

指定的线程池而不是默认的。