# Thinking in java

--三人务于精熟，而亮独观其大略

# 背景

## 为啥要学习这本书

### 1.掌握java基础知识

我们学习java类的开源工程，比如tomcat，总是有一种力不从心的感觉。其实本质原因就是java基础不扎实。

因此，我们计划通过这本书，实践书中的代码，提升java基础能力。

## 2.助力学习开源系统

之前我们在学习Hikari源码的时候，碰到一些java的concurrency相关的类。我们对这些类了解不是很深入，只了解一些大概。这对我们学习java技术栈的开源工程当然是一个阻碍。为了助力我们学习开源工程，我们针对各个专题进行java基础学习。

## 3.解决一些特定的业务场景

本书的一大特色，就是除了提供各种java语法的说明，还提供了一些基于java语法的各种实际业务场景。比如在Concurrency主题下，有一个非常有意思的例子：

|  |
| --- |
| BankTellerSimulation.java |

这就是一个非常贴合银行业务场景：客户排队、柜员处理、柜员经理管理整体的客户队列。以后我们在实际工作中碰到类似的场景，就可以套用书中的代码。

## 本书优点

《Thinking in Java》为啥为这么成功，有哪些优点呢？

### 1.代码非常丰富

基本上每个主题都有代码

### 代码精简、紧凑

能够用最短的例子，把某个Java功能说的清楚；

# Generics

这章主要介绍泛型。在开始学习实践泛型前，我们先了解一下泛型的应用场景。

场景1 集合类

常用的集合类，几乎都用到了泛型，因为集合类中某个元素的类型，是不确定的。比如LinkedList<String>、HashMap<String,String>等等，我们只有在声明一个新的集合类的时候，才会确定这个集合类中元素的类型。

场景2 tomcat

我们学习How Tomcat Works的时候，可以看到很多用到泛型的地方。按照<T>搜索就行了：



## Simple generics

代码

|  |
| --- |
| Holder1.java  LinkedStack.java  RandomList.java  TwoTuple.java |

## Generic interfaces

泛型在interface中的用法，就是在java 集合类中大量的用法：

List<String> list = ...这种

可以在集合类中放各种类型

代码1 CoffeeGenerator.java

(Generics/interface 包下的各个代码)：



整体架构如下：



代码1 这一套代码非常不错，高效、紧凑。后续如果有类似的场景，比如我们有一个对象的池，我们希望有一个generator，能够生成随机对象，就可以用这个模型。

代码2

|  |
| --- |
| Fibonacci.java |

## Generic methods

泛型在方法中的应用

代码1：

|  |
| --- |
| Generators.java |

这个代码的功能：通过Generator.next()创建一个个元素，创建一个长度为n的集合类。这个类通过fill()方法中应用泛型，最大限度保证了Generators的通用性。

这个代码非常有用，后续我们如果要创建一个集合类，就可以用这个方法，两个步骤：

1.定义个Generator类，负责创建集合中一个个元素

2.Generators.java创建结合

代码2

|  |
| --- |
| GenericVars.java |

这个代码是Generics 在methods中的应用场景2，GenericVars的功能为，将array转化为list。泛型的使用，使得这个功能最大限度做到了通用性

## Anonymous inner classes

代码：

|  |
| --- |
| BankTeller.java |

这段代码和银行大厅的场景非常像，很有意思。后续如果有customer-server模式的场景，可以套用这个代码

## Building complex models

代码：

|  |
| --- |
| Store.java |

这段代码非常有意思，后续如果有那种对象层层嵌套的场景，就能套用这个代码。

## The mystery of erasure

## Compensating for erasure

## Bounds

## Wildcards

## Issues

## Self-bounded Types

## Dynamic type safety Exceptions

## Mixins

关于Mixins，我们简单说明一下，意思就是，我们一系列的类，我们希望把这些类都整合到一个统一的类中(称为mixin)，我们希望mixin类包含这些所有类的方法。

### Mixin方案1: 新建一个mixin类

新建一个Mixin类,实现所有要mixin的接口和方法

### Mixin方案2: 装饰类

新建一个装饰类,这个装饰类通过继承的方式,引入了要mixin的方法，然后其他类再通过继承装饰类的方式,间接获取要访问的方法.

### Mixin方案3: 动态代理

通过MixinProxy.newInstance()类,创建一个动态代理类，这个动态代理类包含了所有要mixin的类和对应类的接口，我们可以通过这个动态代理类,访问mixin的所有的类

具体代码参考如下package：

|  |
| --- |
| com.nbcb.thinkingInJava.generics.mixins |

我们比较一下三个防范，方案3显然是最好的，因为这个方法无需提前新建一个类，比如mixin类、或者装饰类什么的。可以随时可以引入各种类。具体做法是，通过动态代理的方式，创建一个动态代理类，这个类通过反射的形式，自动整合了所有需要mixin的对象。最终我们可以通过访问这个动态代理类，访问所有我们想要访问的方法。

## Latent typing

我们开始学习latent type，所谓的latent type，就是”潜在的类型”。其实就是让java代码去猜测，一个对象有哪些方法，然后去执行这个代码。

在这方面，和python/C++相比，java自动猜测对象包含哪些方法的能力，是有所不足的

java中如何说明某个对象包含哪些方法呢？一个典型的场景，就是让某个类实现某个接口，然后我们就能知道这个类必然有哪些方法了。参考代码：

|  |
| --- |
| DogAndRobot.java |

## Compensating for the lack of latent typing

如何解决latent typing的问题呢？

方案1 反射

反射例子1

通过反射的方式，在执行某个对象的时候，去检测这个对象都有哪些方法，如果有我们想要执行的方法，就执行它。参考代码：

|  |
| --- |
| LatentReflection.java |

反射例子2

例子2也是反射，但是可以指定触发对象的某个方法。

参考代码：

|  |
| --- |
| ApplyTest.java |

方案2 Adapter模式

比如我们期望调用对象A的add()方法，但是对象A没有这个方法怎么办呢？方案2的方式是：将我们要操作的对象，通过Adapter模式，创建一个新的对象，这个对象包含了add()方法。

参考代码：

|  |
| --- |
| Fill2.java |

## Using function objects as strategies

代码参考：

|  |
| --- |
| Functional.java |

这个例子非常牛逼，介绍了函数式编程的原理，要好好体会！

实践了这个例子，就能够深入理解函数式编程的原理。这个例子把generic的特性发挥

到了极致，同时，引入了strategies设计模式的原理。

## Summary

### 什么是泛型

我们通过各种代码实践，接触了泛型的功能和用法。从总体来看，泛型<T>就像是一个虚拟的位置，放在某段代码中，在这段代码中<T>中的T是什么类型，其实不是很重要。

比如经典的结合类List<T>，我们在写List接口，或者在写实现类ArrayList的时候，根本不关心List中的元素是什么类型。

再举一个例子，比如在现代企业中，如果一个企业的整个架构良好，规章制度清晰，大家各司其职，那么，公司的CEO换谁来当，都能够保证企业能够稳步向前。这时，CEO的位置，看起来就像是一个泛型<T>。

一个现代国家也是一样，国家元首，其实只是一个象征，一个泛型<T>，用通俗的话来说就是，一个制度设计良好的国家，换谁来当元首都一样。

可能这和我们固有的创业思路有所不同。我们在创业初期，往往雄心勃勃，有浓烈的个人英雄主义。但是当公司规模慢慢扩大的时候，如果公司的正常运转还是要依赖某个人，或者某几个人，那么这个公司可能就不是好的公司。

所以，在业务发展的同时，如何依托现代化的企业管理学，构建起一个合理的架构，就变得非常重要。

### 本章的收获

学习本章可以说是收获满满。不仅能够深入了解泛型的使用场景，还顺带引入了各种设计模式、函数式编程的原理。作者的奇思妙想、宽广的视野、对原理的深入理解，令人叹为观众。

所以我们经常说，Java语言本身虽然仅仅只是一门语言，但是以Java语言为核心，构建出一整个生态。

### 3.每个主题之间互不关联。

很少存在先要学习示例代码1，才能学习示例代码2的问题。有是有，但是整体来说关联度不是很大。

### 4.有很多优秀的解决方案

我们看到书中的示例代码，除了用于说明各个章节的知识点，还给我们提供了很多有用的解决方案，你可以称之为”design pattern”(设计模式)，也可以成为”tool box”(工具箱)。总之，你可以在书中找到很多有用的代码，用到我们实际的项目工作中去。

比如在generics章节中，可以通过Generators.fill()方法，创建一个集合，这个集合中的元素类型是可以指定的，元素生成的方式是可编程的，这就是一个非常有用的解决方案。

## 本书代码

本书代码在官网上就有，下载到本地：

/Users/zhoushuo/Documents/Tech/thinking in java

# Inner Classes

# Holding your objects

# Error Handling with Exception

# Strings

String这块相关的代码包括：

|  |
| --- |
| Immutable.java  Concatenation.java |

# Type Information

通过反射的原理，了解java类的类型

# Arrays

本章的代码如下：

|  |
| --- |
| ContianerComparision.java  IceCream.java  CountingGenerator.java  RandomGenerator.java  Generated.java  ConvertTo.java  CopyingArrays.java  ComparingArrays.java  CompType.java  Reverse.java  CompTypeComparator.java  StringSorting.java  ArraySearching.java  AlphabeticSearch.java |

## Create test data

这个小节非常有意思，结合generics，创建各种类型的数组，包含如下代码：

|  |
| --- |
| CountingGenerator.java  RandomGenerator.java  Generated.java  ConvertTo.java |

## Array Utilities

这个小节介绍array 各种utilities方法，包含如下代码：

|  |
| --- |
| CopyingArrays.java  ComparingArrays.java  CompType.java  StringSorting.java  ArraySearching.java  AlphabeticSearch.java |

# Containers in Depth

深入了解集合类

# I/O

IO相关的代码包括：

|  |
| --- |
| DirList.java  DirList2.java  DirList3.java  Directory.java  PPrint.java  ProcessFiles.java  MakingDirectories.java  BufferedInputFiles.java  MemoryInput.java  FileOutputShortcut.java  StoringAndRecoveringData.java  UsingRandomAccessFile.java  TextFile.java  BinaryFile.java  Echo.java  OSExecute.java  GetChannel.java  BufferToText.java  AvailableCharSet.java  IntBufferDemo.java  ViewBuffer.java  UsingBuffers.java  LargeMappedFiles.java  MappedIO.java  LockingMappedFiles.java  FileLocking.java  GZIPcompress.java  ZipCompress.jva  Worm.java  Alien.java  FreezeAlien.java  ThawAlient.java  Blips.java  Logon.java  SerialCtl.java  MyWorld.java  StoreCADState.java  RecoverCADState.java  People.java  Person.java  PreferenceDemo.java |

可以看到，整体IO相关的代码还是非常多的。涉及非常多的内容。我们分门别类实践一下各个demo。

## The File Class

|  |
| --- |
| DirList.java  DirList2.java  DirList3.java  Directory.java  PPrint.java  ProcessFiles.java  MakingDirectories.java  BufferedInputFiles.java |

其中，DirList.java/DirList2.java/DirList3.java三个代码都是说明如何通过File.list()方法,列出某个目录下的所有文件/目录，分别通过定义DirFilter实现类、内部类、匿名内部类的方式，实现Filename的正则表达式过滤。

## Input and Output

## Readers and Writers

## I/O Streams

相关代码包括：

|  |
| --- |
| 以下代码作为一组，说明如何读取file  BufferedInputFile.java  MemoryInput.java  FormattedMemoryInput.java  TestEOF.java  输出file  BasicFileOutput.java  读写文件  StoringAndRestoringData.java  UsingRandomAccessFile.java |

## File reading and writing utilities

代码包括：

|  |
| --- |
| TextFile.java |

## Standard IO

代码包括：

|  |
| --- |
| Echo.java  ChangeSystemOut.java  Redirecting.java |

## Process control

代码包括：

|  |
| --- |
| OSExecute.java |

## New IO

代码包括：

|  |
| --- |
| GetChannel.java  BufferToText.java  ChannelCopy.java  AvailableCharSet.java  TransferTo.java  GetData.java  IntBufferDemo.java  ViewBuffer.java  UsingBuffers.java  LargeMappedFiles.java  MappedIO.java  LockingMappedFiles.java  FileLocking.java |

New IO相关的代码还是挺多的

## Compression

代码包括：

|  |
| --- |
| GZIPcompress.java  ZipCompress.java |

这两个代码包含了单个文件、多个文件的压缩、解压

## Object Serialization

## IO总结

从整体来说，IO部分的代码还是非常有用的。在工作中也是强相关的，比如input/output stream的处理、Compress、序列化/反序列化，在影像平台还是有非常广泛应用。

另外，在tomcat中IO也有非常重要的作用，比如对于content的处理，就涉及到IO。

本章不仅涉及了传统的stream，还涉及了NIO，大大提升了IO读写效率，降低了内存使用率。

后续，我们可以找机会把NIO这些内容应用到工作中去，提升文件处理的效率。

当然，我们这章是基于JDK1.5的，后续JDK1.8对于stream的处理，又有了新的提升，这块内容也要不断学习补充。

# Enumerated Types

# Annotation

## 背景

我们在开发影像新的后台服务的时候，开始大量接触到了annotation。在实践过程中，annotation确实对于提升代码的可读性，有非常大的帮助。在SpringBoot体系中，注解也是非常重要的一个组成部分。

因此，我们在这里专题学习注解。主要目标是：1.了解注解的实现原理；2.能够自己实现注解；3.了解SpringBoot体系中核心注解的实现原理。

《Thinking In Java》中，Annotation相关的内容有些已经过时了，建议参考最新的《On Java 8》，针对JDK8，对内容进行了更新。尤其是把APT改为了javac，需要特别注意。

## Writing annotation processors

相关代码1：

|  |
| --- |
| UseCaseTracker.java  UseCase.java  TableCreator.java  PasswordUtil.java |

相关代码2：

|  |
| --- |
| DBTable.java  Constraints.java  SQLString.java  SQLInteger.java  Member.java  TableCreator.java |

这一组代码主要是通过annotations方式，将一个java bean转化为SQL DDL语句。

具体实现方式是这样的：

1. 定义一系列的annotation

这些annotation用来标识java bean中定义的表名、字段类型等

1. 通过反射机制读取java bean
2. 解析java bean中的annotation
3. 根据annotation，将java bean中的字段转化为SQL DDL语句

## Using Javac to process annotations

## JUnit

从实际使用过程中，JUnit还是有很多限制的，比如constructor只能有一个，并且不能带有参数，这是什么鬼？我们加入JUnit测试后，难道还要我们修改主工程的源码？

JUnit提供的注解其实也不多，核心注解就只有@Test。只能说是够用。

但是，这个小结的精髓并不是告诉你JUnit怎么怎么样，而是告诉你，像是JUnit这样的框架，是如何实现的。官网代码就实现了这些注解。我们可以参考一下。

相关代码：

|  |
| --- |
| AtUnitComposition  AtUnitExample1.java  AtUnitExample2.java  AtUnitExample3.java  AtUnitExample4.java  AtUnitExample5.java  AtUnitExternalTest.java  HashSetTest.java |

## Annotation的应用

Annotation的应用当然是非常广泛的，我们可以把Annotation应用到我们的自动化测试框架TestUI上去。

## 总结

# Concurrency

## Basic Threading

1.

CallableDemo.java

说明如何从线程获取返回值

2.

SleepingTask.java

说明线程休眠一段时间

3.

SimplePriorities.java

说明如何设置线程优先级(貌似设置一下优先级也没啥用)

4.

ResponsiveUI.java

说明线程在交互式UI中的应用

## **Sharing resources**

1.

通过一个偶数生成器，来判断判断线程是否安全

EvenChecker.java

IntGenerator.java

线程不安全的偶数生成器

EvenGenerator.java

线程安全的偶数生成器

MutexEvenGenerator.java

SynchronizedEvenGenerator.java

2.

序列号生成器(通过优化，保证序列号生成器线程安全)

SerialNumberChecker.java

SerialNumberGenerator

AtomicIntegerTest.java

### 3.CriticalSections

CriticalSection的概念

CriticalSection.java

这个代码要好好总结一下，尤其是PairChecker，通过异步线程检测数据是否一致，后续可以参考

SyncObject.java

### 4.Thread local storage

说明Thread local storage的应用，说白了就是把共享资源在各个线程中各存放一份，以免竞争

ThreadLocalVariableHolder.java

### 5.Sharing resource总结：

这个小结主要介绍了集中方案，用于避免线程不安全导致数据异常：

1.synchronize关键字

2.try lock

3.Java自带的各个Atomic类

4.critical section(缩小synchronize范围)

5.Thread local

## **Terminating tasks**

1.

说明各个情况下的线程能否被中断

Interrupting.java

2.

如何关闭子线程中的资源

CloseResource.java

3.

NIO的方式，只要关闭子线程，就能够自动将子线程中的NIO 连接资源关闭

NIOInterruption.java

4.

MultiLock.java

5.

这个代码为了说明*Lock.lockInterruptibly()*的用法

Interrupting2.java

6.

这个代码主要是为了说明，我们中断一个子线程的时候，需要特别注意关闭子线程中的资源

InterruptingIdiom.java

Terminating tasks 《总结》

1、2、3是一个系列，层层递进，说明了关闭子线程的时候，如何释放将子线程中各种类型的资源。

## **Cooperating between tasks**

### wait() and notifyAll()

这个代码主要是借助汽车上蜡、擦拭这两个独立的线程，说明线程间如何通过wait()/notifyAll()进行通信。

关键字是wait()/notifyAll()

代码示例：

|  |
| --- |
| WaxOMatic.java |

### 2.NotifyVsNotifyAll

notify()/notifyAll()的区别在于，如果多个线程，共享一个static对象，然后这些线程都wait on the static object，这时，如果调用notify()，就只会通知一个线程；如果调用notifyAll()，会通知所有线程。从整体来看，notify()/notifyAll()的区别在于通知线程的范围有所不同

代码示例：

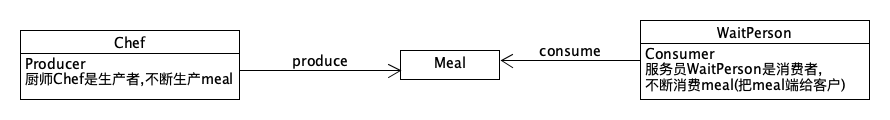
|  |
| --- |
| NotifyVsNotifyAll.java |

### 3.Producer and Consumers

主要是通过生产者、消费者模型，说明进程间是如何通信的，关键字是notifyAll()

|  |
| --- |
| Restaurant.java  WaxOMatic2.java |

其中Restaurant.java的生产者、消费者场景如下：



### 3.Producer Consumers and Queue

所谓的BlockingQueue，其实就是一个支持并发访问的队列，提供一些take()/put()等方法，往queue中存取元素。  
  
\* BlockingQueue接口的具体实现类有：  
\* 1.LinkedBlockingQueue  
\* 2.ArrayBlockingQueue  
\* 3.SynchronousQueue

实例代码：

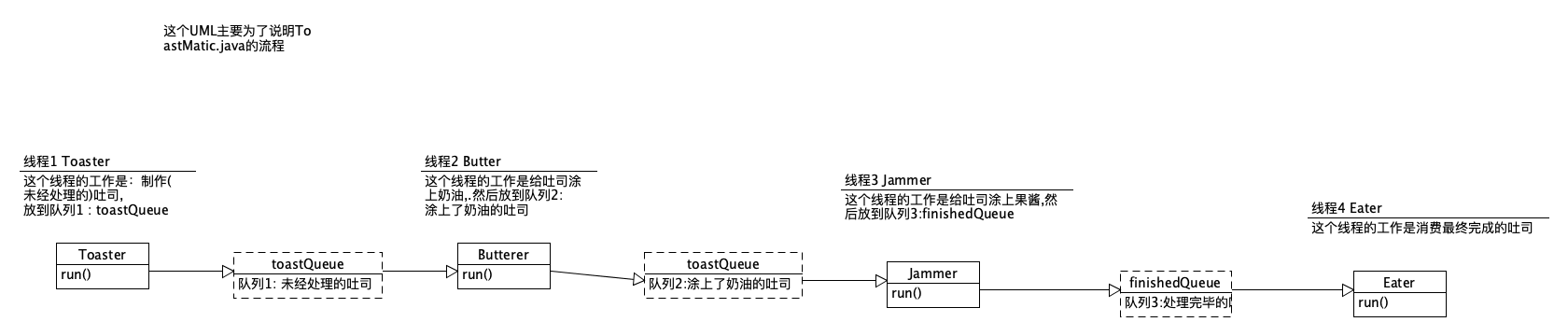
|  |
| --- |
| TestBlockingQueues.java |

下面这个代码的核心也是BlockingQueue这个适应并发场景的数据结构，在TestBlockingQueues.java的基础上更近一步，TestBlockingQueues还是以BlockingQueue语法为主，业务场景不多，本代码就是基于一个非常有趣的业务场景

示例代码：

|  |
| --- |
| ToastOMatic.java |

整体流程如下：参考UML/concurrency-ToastMaitc.uxf

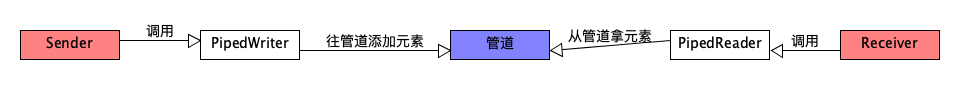


### 4.Using pipes for I/O between tasks

这里例子引入了一个新的类： PipedWriter。光从名字就能看出是一个管道，用于并发线程之间传输内容，功能和之前的BlockingQueue差不多，就是安排了一个队列，一部分线程往队列中添加元素，一部分线程从队列中拿元素。

代码:

|  |
| --- |
| PipedIO.java |



## **Deadlock**

这个package下的代码，说明了死锁的的场景(DeadlockingDiningPhilosopher.java)，

以及如何解决死锁(FixedDiningPhilosopher.java)。

|  |
| --- |
| DeadlockingDiningPhilosopher.java  FixedDiningPhilosopher.java |

代码示例就是非常典型的哲学家模型，思路非常清晰

|  |
| --- |
| // 哲学家类  Philosopher.java  // 会造成死锁的场景  DeadlockingDiningPhilosopher.java  // 修复了死锁的场景  FixedDiningPhilosopher.java |

## **New Lib**

### CountDownLatch

// 这个文件主要是为了说明*CountDownLatch*这个用于并发的类库

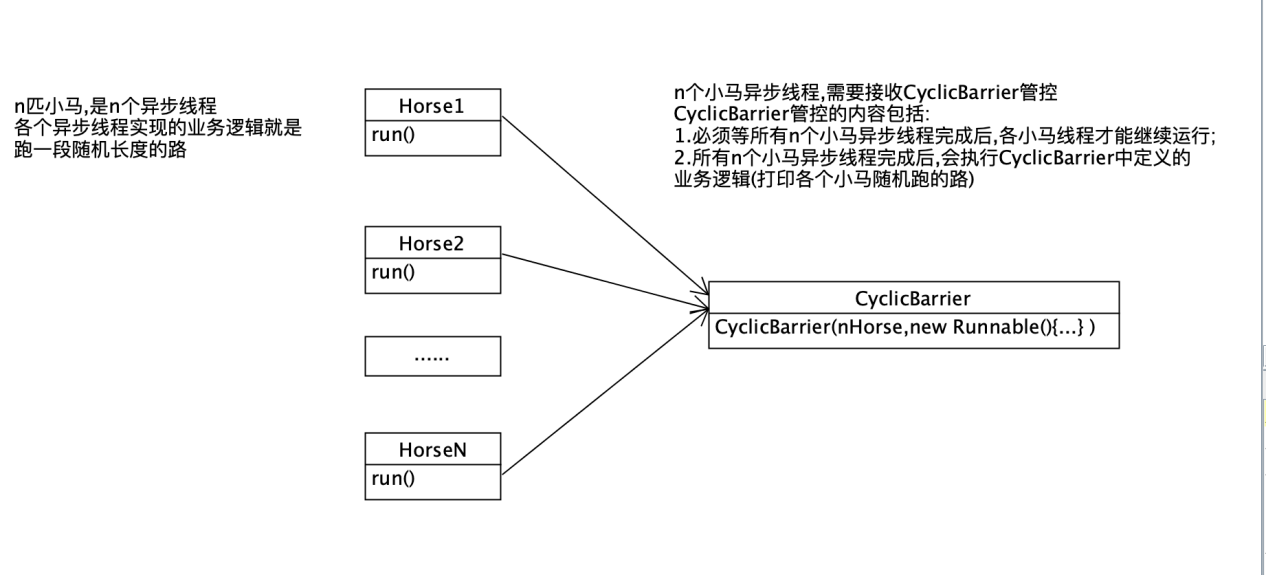
CountDownLatchDemo.java

### CyclicBarrier

赛马的场景，主要是为了说明 CyclicBarrier 的用法。CyclicBarrier的功能正如其字面意义一样,循环屏障。屏障的意思是CyclicBarrier会挡住所有调用CyclicBarrier.await()方法的异步线程，必须等到这些异步线程全部完成，屏障才会消失，程序才会往下跑。循环的意思是CyclicBarrier能够重复使用。

CyclicBarrier其实和CountDownLatch差不多，就是很多线程在跑，但是所有的线程要等待CyclicBarrier中定义的动作，只有等待CyclicBarrier中定义的动作完成之后，各个线程才能开始继续跑。区别在于，CyclicBarrier可以重复使用。具体可以看下面这个赛马的例子，非常有趣：

|  |
| --- |
| HorseRace.java |



### DelayQueue

这又是一个非常有意思的例子，引入了一个新的类：DelayQueue  
\* DelayQueue特点是这样的：  
1.队列中的对象必须要实现Delay接口  
 这个Delay接口用来设置delay参数，某个对象delay设置得越大，  
 说明这个对象越晚从队列中被获取  
2.producer队列可以往DelayQueue添加各个元素，  
 通过Delay接口指定各个元素多久时间以后能够被获取  
3.consumer接口可以从DelayQueue对了获取各个元素  
  
总结一下：  
1.DelayQueue整体来说也是用于并发场景：producer-consumer  
2.可以指定队列中的各个元素的优先级

|  |
| --- |
| DelayQueueDemo.java |

### PriorityBlockingQueue

本代码沿袭了上一个代码：DelayQueueDemo，不能说一模一样，但是相似度那是99%。在DelayQueueDemo中，决定一个任务的优先是delay，就是等待时间越久的元素，越后面才被取到。  
  
 PriorityBlockingQueue意思就是可以设置队列中各个元素的优先级：哪个元素的优先级越高，就越先被取到。当然PriorityBlockingQueue也是用于并发场景的：

1. Producer并发往PriorityBlockingQueue中添加一个个指定优先级的元素；
2. Consumer从PriorityBlockingQueue中获取元素，这些元素是按照优先级顺序一个个被consumer取走的。

|  |
| --- |
| PriorityBlockingQueueDemo.java |

### Greenhouse Controller with ScheduledExecutor

\* 这个代码的核心就是ScheduledThreadPoolExecutor  
\* ScheduledThreadPoolExecutor能够实现两个功能：  
\* 1.启动一个异步线程  
\* schedule(task, delay, TimeUnit.MILLISECONDS)  
\*   
\* 2.定期轮循一个异步线程  
\* scheduler.scheduleAtFixedRate(task, initialDelay, period, TimeUnit.MILLISECONDS);  
\*   
\* @业务场景  
\* 这个代码的业务场景是这样的，一个阳光房，定期实现：  
\* 1.定期打开、关闭灯光  
\* 2.定期打开、关闭水源  
\* 3.定期调整恒温模式  
\* 4.定期打铃  
\* 5.定期采集数据

代码示例：

|  |
| --- |
| GreenhouseScheduler.java |

### Semaphore

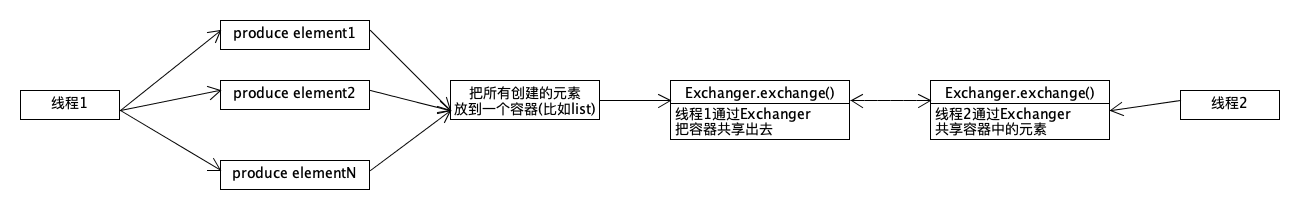
下面这三个文件，主要是通过实现一个object pool，验证Semaphore的功能

|  |
| --- |
| Pool.java  Fat.java  SemaphoreDemo.java |

### Exchanger

ExchangerDemo这个例子为了说明Exchanger在并发编程中的用法。并发编程中有一个场景，就是线程1中会创建若干对象，然后希望把这些对象共享给其他线程。Exchanger可以实现这个功能，不仅能够实现对象在线程间的共享，还能够避免各种并发问题。

|  |
| --- |
| ExchangerDemo.java |

大致流程如下：

### New lib总结

我们实践了new lib下的各个主题，总结一下。这个小节中提供了很多语法糖，用于并发场景下的各种问题：

|  |  |
| --- | --- |
|  | 说明 |
| CountDownLatch | CountDownLatch我们之前已经实践了很多了，说白了就是一个用于控制异步线程的计数器，比如某个异步线程结束后，计数器减1，当所有异步线程任务完成后，计数器变为0。 |
| CyclicBarrier | 正如CyclicBarrier这个名字所示(周期性的障碍)，意思很明确，就是会造成各个异步线程周期性挂起。 |
| DelayQueue | DelayQueue的意思也很清晰，就是一个队列，我们可以设置队列中的各个元素的delay值，通过这个delay值调整各个元素的优先级，优先级高的元素优先被取出来 |
| PriorityBlockingQueue | PriorityBlockingQueue也是一种队列，可以设置各个元素优先级。这次不用delay设置优先级了，这次各个元素通过实现compareTo()接口方法设置优先级。 |
| Greenhouse | Greenhouse的核心其实就是ScheduledThreadPoolExecutor，可以实现启动一个异步线程，或者定期定期轮循一个异步线程 |
| Semaphore | 所谓的Semaphore，就是信号量。通过调用Semaphore.acquire()/release()方法，解决并发线程访问共享资源的问题。 |
| Exchanger | Exchanger主要解决的场景，就是两个异步线程之间，要交换某些对象。 |

## **Simulation**

### BankTeller Simulation

这个代码，从总体来说就是模拟并发场景：银行柜员处理客户业务.有一个客户队列(CustomerLine)，队列中有若干个客户(Customer)在排队,然后有若干个柜员(Teller)在处理客户业务.同时，有一个运营经理(TellerManager)，针对当前客户队列的长度，对柜员的人数进行调整.后续优化：Teller的数量是固定的，不可能根据客户数量，无限增长

|  |
| --- |
| BankTellerSimulation.java |

### Restaurant Simulation

示例代码：

|  |
| --- |
| RestaurantWithQueues.java |

这应该是目前为止本书最复杂的一个代码了。引入了Customer/WaitPerson/Chef/Restaurant/Order/Plate这些对象

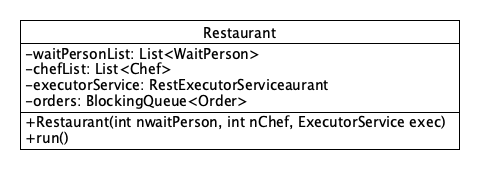
1. **异步线程**

其中Customer/WaitPerson/Chef/Restaurant都实现了Runnable接口，是异步线程类。这个代码就是讲这些异步线程之间怎么交互。

1. **BlockingQueue**

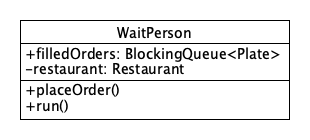
值得注意的是，代码中有2个BlockingQueue:

1. -orders: BlockingQueue<Order>



这是客户下的订单，放在这个队列中，这个队列保存在Restaurant对象中。这个队列用于WaitPerson和Chef这两个异步线程之间的交互。

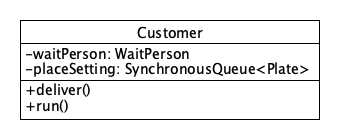
(2)+filledOrders: BlockingQueue<Plate>



厨师做完的食物，保存在这个队列中，这个队列保存在WaitPerson对象中。这个队列也是用于WaitPerson和Chef这两个异步线程之间的交互。

正如书本在这个小节结尾的时候所说，这个代码的核心就是两个2个BlockingQueue，通过这2个队列，解决了不同的异步线程之间的交互问题。以后我们面临类似的场景的时候(异步线程之间的通信)，就可以用这个思路。当然，在实践中，我们一般通过把待处理任务持久化(放到数据库)，实现异步线程的数据交换。

3.**SynchronousQueue**

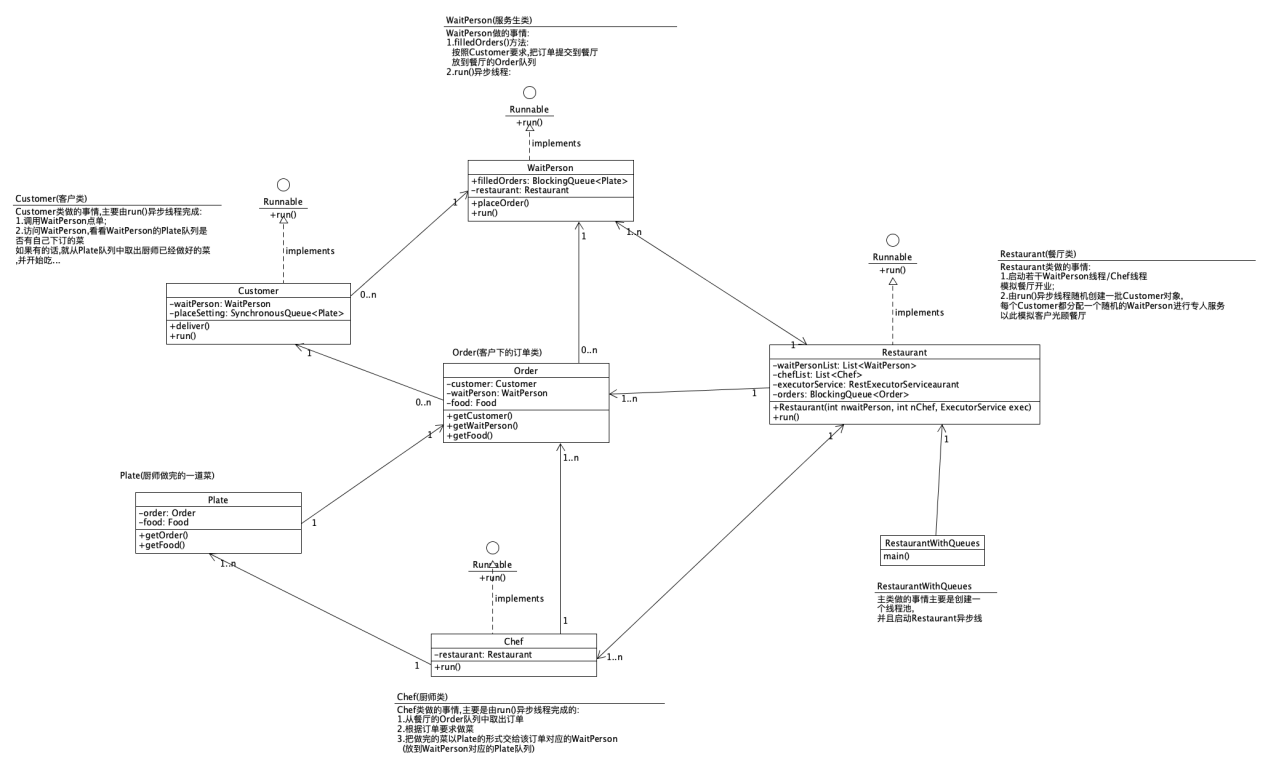


-placeSetting: SynchronousQueue<Plate>

这是客户从WaitPerson拿到做好的事务，放到这个队列中。

SynchronousQueue是这个代码里新出现的，作用请参考Customer类中的注释，实现的效果就是保证两个异步线程(比如Producer/Consumer)之间的单个元素交换。

以下是整体的UML图：



当然，最后不要忘了首尾工作。当我们关闭线程池之后(模拟饭店关门)：

1. Customer是否还有下的单还未完成；
2. WaitPerson手中是否还有做完的菜还没端给Customer；
3. Chef手中是否还要正要做，但是没有完成的菜；
4. Restaraunt是否有还未完成的订单；

我们后续要研究一下，如何使得程序更加优雅地退出。

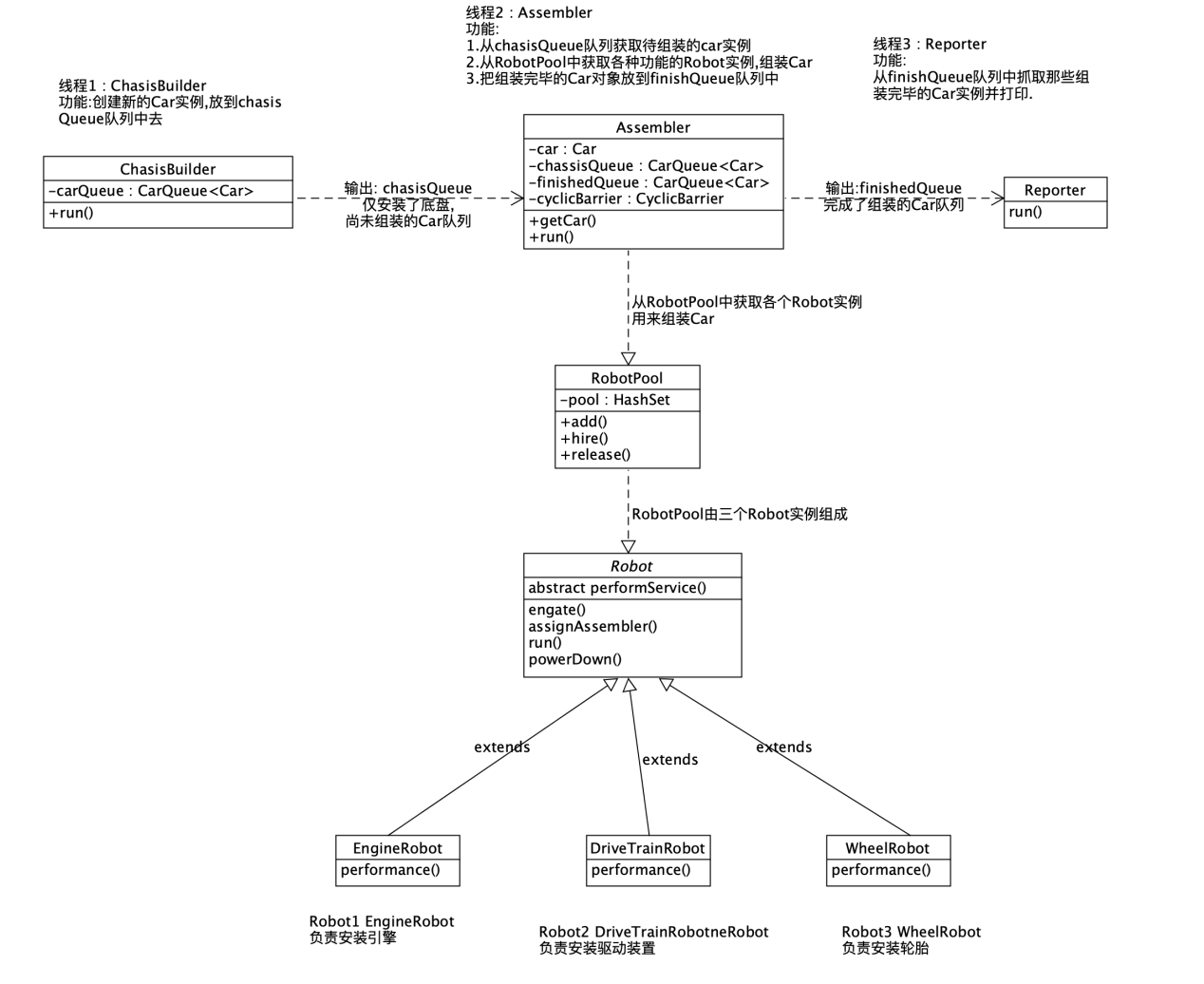
总结一下，我们都知道，并发编程不是一件容易的事情，尤其是这个例子。这样的场景，如果靠自己从头编码，那是不现实的。所以我们要把Concurrency这章的各个示例代码认真实践一遍。以后碰到并发场景，直接套用书中的示例代码就行了。

### Distrubuting work

\* 这个代码综合了Concurrency的各种技术，涉及：  
\* 1.BlockingQueue 保存各种Car队列；  
\* 2.CyclicBarrier 用于协调各个异步线程；  
\* 3.wait()/notifyAll() 用于Robot异步线程的状态激活、断电  
\*  
业务场景  
\* 这个代码的业务场景就是汽车组装  
\* 组装流程如下：  
\* 1.通过ChasisBuilder创建一个Car队列:chasisQueue，  
\* chasisQueue队列中的Car仅仅只有底盘，未开始组装；  
\* 2.Assembler从chasisQueue队列中获取待组装的Car对象  
\* 然后从RobotPool中获取三个Robot实例，  
\* 分别组装Car的各个部件(引擎、驱动装置和轮胎)  
\* 将组装完毕的Car放到finishedQueue队列  
\* 3.Reporter从finishedQueue队列获取那些组装完毕的汽车，  
\* 打印这些汽车的信息

|  |
| --- |
| CarBuilder.java |

**整体架构的UML**



代码整体还是比较复杂的，

整体流程有点类似流水线：

**流水线节点1**： ChasisBuilder

ChasisBuilder负责创建新的Car实例,放到chasisQueue队列中去

**流水线节点2**： Assembler

Assembler负责:

1.从chasisQueue队列获取待组装的car实例

2.从RobotPool中获取各种功能的Robot实例,组装Car

3.把组装完毕的Car对象放到finishQueue队列中

**流水线节点3** ：Reporter

Reporter负责从finishQueue队列中抓取那些组装完毕的Car实例并打印.

最关键的技术点有两个：

1. 通过CyclicBarrier实现异步线程的协调；
2. 通过wait()/notifyAll()实现对象池中各个对象状态的变化；

**优化内容**：

我们基于示例代码，进一步做了优化：

1.thread pool shutdown 之后，关闭整个流水线中各个线程更加优化：避免了异常堆栈打印，能够打印处于assembler(正在组装中)的Car实例：

Car{id=17, engine=true, driveTrain=true, wheel=false}is in assembler while assembler BarrierException broken ...

这个意思就是id为17的Car实例，完成了engine组装，但是传动装置和轮胎还未组装，这时我们如果通过关闭线程池，关闭整个流水线的话，就会打印这个正处于组装状态的Car；

2.thread pool shutdown 之后，能够打印那些等待安装的Car

当然，后续好友更多的优化，比如，如何在关闭之前，把那些处于chasisQueue中、待组装的机器组装完，再关闭整个流水线。这样关闭更加优雅。

## **Performance Tuning**

### Comparing mutex technology

这是一个简单的，用来评测线程之间同步效率的程序,主要评测的对象是两种同步方案：  
\* 1.synchronized  
\* 2.Lock

实际测试下来效率差不多

|  |
| --- |
| SimpleMicroBenchmark.javas |

|  |
| --- |
| SynchronizationComparisons.java |

### Lock-free containers

### Optimistic locking

### ReadWriteLocks

Tester.java

MapComparisons.java

FastSimulation.java

ReaderWriterList.java

ActiveObjectDemo.java

## **总结**

Java的并发还是有很多内容可以讲的。这章，不仅介绍了各种并发相关的语法，还介绍相关的并发业务模型，非常有用。当然，java并发的内容博大精深，还有很多扩展知识需要学习。在开源项目中，保证并发性能，也是非常重要的一块。

我们看到并发编程还是非常复杂的，仅仅只是看这本书，仅仅只是停留在理论阶段，就觉得很复杂，更不要说放到实际生产环境了。因此，我们尽量要套用书中的一些场景，不要自己去实现底层的notify()/wait()这些，否则极易出现各种并发问题。