# 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

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。

## 本书优点

《Thinking in Java》为啥为这么成功，有哪些优点呢？

0.代码非常丰富

基本上每个主题都有代码

1.能够用最短的例子，把某个Java功能说的清楚；

2.每个主题之间互不关联。不存在先要学习示例代码1，才能学习示例代码2的问题。

## 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

# 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

## 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：

|  |
| --- |
| 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

## Anonymous inner classes

## Building complex models

## Compensating for erasure

# 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.

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.

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**

1.这个代码主要是借助汽车上蜡、擦拭这两个独立的线程，说明线程间如何通信

关键字是*notifyAll()*

WaxOMatic.java

2.

NotifyVsNotifyAll.java

3.

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

TestBlockingQueues.java

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)。

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**

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

CountDownLatchDemo.java

// 赛马的场景，主要是为了说明 *CyclicBarrier* 的用法

HorseRace.java

DelayQueueDemo.java

PriorityBlockingQueueDemo.java

GreenhouseScheduler.java

// 下面三个文件，主要是通过实现一个object pool，验证Semaphore的功能

Pool.java

Fat.java

SemaphoreDemo.java

ExchangerDemo.java

## **Simulation**

BankTellerSimulation.java

RestaurantWithQueues.java

CarBuilder.java

## **Performance Tuning**

SimpleMicroBenchmark.java

SynchronizationComparisons.java

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。各种并发模型