# 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.

## 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；

### 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

# Type Information

通过反射的原理，了解java类的类型

# 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。

# Arrays

# Containers in Depth

深入了解集合类

# I/O

# 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.

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