java并发编程笔记—Immutable Object模式

|  |  |
| --- | --- |
|  | 🢂 内容概览 |
|  | Why：此文档用来做什么？它存在的意义是什么？为解决什么问题？   |  | | --- | |  |   What：当前包含了那些内容？   |  | | --- | |  |   How：此文档应如何参考？   |  | | --- | |  |   Who：此文档适用于那些人员阅读参考？   |  | | --- | |  |   Summary：摘要   |  | | --- | | Immutable  CopyOnWrite  读多写少，读写分离 |   Reference：参考文献   |  | | --- | | [Java多线程系列--“JUC集合”02之 CopyOnWriteArrayList](http://www.cnblogs.com/skywang12345/p/3498483.html)  [Java多线程系列--“JUC集合”03之 CopyOnWriteArraySet](http://www.cnblogs.com/skywang12345/p/3498497.html)  [Redis 复制原理及分析](http://my.oschina.net/hanruikai/blog/308007?p=1)  [Redis内存存储结构](http://blog.csdn.net/iloveyin/article/details/7481447) | |

目录

[1 模式描述 4](#_Toc461095358)

[1.1 不可变对象 4](#_Toc461095359)

[1.2 Immutable Object模式 4](#_Toc461095360)

[1.3 不可变对象的操作 4](#_Toc461095361)

[1.4 不可变对象的实现约束 4](#_Toc461095362)

[2 相关原则 4](#_Toc461095363)

[2.1 遵循原则 4](#_Toc461095364)

[2.2 违反原则 4](#_Toc461095365)

[3 模式类图 4](#_Toc461095366)

[4 模式实现 5](#_Toc461095367)

[4.1 定义不可变对象：ImmutableObject 5](#_Toc461095368)

[4.2 定义操作对象：Mainipulator 6](#_Toc461095369)

[4.3 客户端访问：Client 6](#_Toc461095370)

[5 适用场景 7](#_Toc461095371)

[5.1 读多写少 7](#_Toc461095372)

[5.2 确保写操作原子性 7](#_Toc461095373)

[5.3 作为缓存的Key 7](#_Toc461095374)

[6 缺点&权衡点 7](#_Toc461095375)

[6.1 对象的copy/new操作消耗性能 7](#_Toc461095376)

[6.2 状态维护困难 7](#_Toc461095377)

[6.3 访问对象带来的安全隐患 7](#_Toc461095378)

[7 应用案例 7](#_Toc461095379)

[7.1 CopyOnWriteArrayList 8](#_Toc461095380)

[7.2 CopyOnWriteArraySet 8](#_Toc461095381)

[7.3 Redis的RDB快照原理 8](#_Toc461095382)

[7.4 Redis的AOF Rewrite原理 8](#_Toc461095383)

[7.5 Redis的数据存储增量Rehash 9](#_Toc461095384)

[8 相关模式 10](#_Toc461095385)

[8.1 Thread Specific Storage模式 10](#_Toc461095386)

[8.2 Serial Thread Confinement模式 10](#_Toc461095387)

[9 问题思考 10](#_Toc461095388)

# 模式描述

## 不可变对象

对象一经创建，其对外可见的状态就保持不变，即对象的属性只能访问不能修改。

## Immutable Object模式

将现实世界中状态可变的实体抽象为状态不可变对象，通过创建不同状态的不可变对象来表示现实世界实体状态的变更。CopyOnWrite的思路可以理解成不可变模式。

## 不可变对象的操作

1. **获取对象的某个状态值：**比如：通过getter方法获取对象某一个属性值；
2. **获取对象的一组状态的快照：**比如：clone一个与当前对象相同状态的对象并返回，需要对对象状态做只读返回，防止引用外泄导致对象状态被外界改变；
3. **生成新的不可变对象实例：**当对象状态需要改变时，创建新的不可变对象来表示这种变化；

## 不可变对象的实现约束

1. **类/方法需要被final修饰：**防止子类通过继承重写对象的访问方法，从而修改对象状态；
2. **类中所有字段都使用final修饰：**一方面约束字段不可被改变；另一方面，通过JMM机制保证属性在初始化后才会被其它线程可见；
3. **防止属性引用外泄：**如果类中的属性引用了其它可变对象，则该属性一定是private的，并且引用不能外泄；同时，在访问时，都需要进行防御性复制，防止属性的引用外泄；
4. **防止This关键字外泄：**对于不可变对象，尽可能避免定义内部类、匿名类，防止内部类和匿名类对对象状态的更改；

# 相关原则

## 遵循原则

**最小知识原则：**尽可能减少不必要的状态暴露给外界；尽可能的减少对象状态的更改操作；

## 违反原则

# 模式类图

|  |
| --- |
|  |

**模式角色如下：**

1. **ImmutableObject（不可变对象）：**状态不可改变的对象，存放状态数据；
2. **Manipulator（不可变对象操作类）：**用于维护不可变对象状态的操作类，当需要对不可变对象进行修改时，通过新建不可变对象的方式实现状态的变更；
3. **Client（客户端）：**访问不可变对象的客户端；

# 模式实现

## 定义不可变对象：ImmutableObject

|  |
| --- |
| **package** com.fsindustry.cime.dpattern.immutableObject.sample;  */\*\*  \* <h1>不可变对象</h1>  \* 类标识为final,防止子类通过继承,对对象状态进行修改  \*/* **public final class** ImmutableObject {   */\*\*  \* 所有字段标识为final,防止对象初始化后被修改,同时确保字段在初始化后才对外可见;  \*/* **private final** Integer **stateX**;  **private final** Integer **stateN**;   */\*\*  \* <h2>提供有参构造器,用于在对象创建时初始化自己的状态</h2>  \*/* **public** ImmutableObject(Integer stateX, Integer stateN) {  **this**.**stateX** = stateX;  **this**.**stateN** = stateN;  }   */\*\*  \* <h2>使用带防御性复制的getter方法,防止对象状态的引用泄露导致状态被修改</h2>  \*/* **public** Integer getStateX() {  **return new** Integer(**stateX**);  }   **public** Integer getStateN() {  **return new** Integer(**stateN**);  } } |

## 定义操作对象：Mainipulator

|  |
| --- |
| **package** com.fsindustry.cime.dpattern.immutableObject.sample;  */\*\*  \* <h1>对象状态操作器:维护不可变对象状态</h1>  \*/* **public class** Mainipulator {   */\*\*  \* 待修改的不可变对象,使用volatile确保变量的可见性  \*/* **private volatile** ImmutableObject **immutableObject**;   */\*\*  \* <h2>用于初始化不可变对象状态</h2>  \*/* **public** Mainipulator(ImmutableObject immutableObject) {  **this**.**immutableObject** = immutableObject;  }   */\*\*  \* <h2>重建不可变对象,实现对状态的修改</h2>  \*/* **public void** changeStateTo(Integer newStateX, Integer newStateN) {  *//重建不可变对象,改变引用指向* ImmutableObject newObject = **new** ImmutableObject(newStateX, newStateN);  *//改变引用为原子操作,故不需要加锁确保原子性* **this**.**immutableObject** = newObject;  }   */\*\*  \* <h2>访问不可变对象</h2>  \*/* **public** ImmutableObject getState() {  **return immutableObject**;  } } |

## 客户端访问：Client

|  |
| --- |
| **package** com.fsindustry.cime.dpattern.immutableObject.sample;  */\*\*  \* <h1>访问不可变对象的客户端对象</h1>  \*/* **public class** Client {   */\*\*  \* 程序入口  \*/* **public static void** main(String[] args) {   Mainipulator mainipulator = **new** Mainipulator(**new** ImmutableObject(0, 0));  ImmutableObject immutableObject = mainipulator.getState();  System.***out***.println(**"stateX:"** + immutableObject.getStateX() + **" , stateN:"** + immutableObject.getStateN());  mainipulator.changeStateTo(1, 1);  immutableObject = mainipulator.getState();  System.***out***.println(**"stateX:"** + immutableObject.getStateX() + **" , stateN:"** + immutableObject.getStateN());  } } |

# 适用场景

## 读多写少

频繁读不频繁写的对象，可以实现为不可变对象，避免加锁带来的开销。

## 确保写操作原子性

同时对一组相关的数据进行写操作，如果需要保证原子性，可以通过将写操作涉及的一组状态封装成不可变对象，保证原子性。

## 作为缓存的Key

可以使用不可变对象作为KV缓存的key（比如：HashMap），防止key改变导致数据无法正确访问。

# 缺点&权衡点

## 对象的copy/new操作消耗性能

不可变对象的每次写操作就要进行一次copy/new操作，带来的性能消耗随着copy的数据量显著增加，包括内存的消耗以及copy/new过程的时间消耗；故不适合copy/new数据量很大，并且写操作很多的场景。

## 状态维护困难

现实世界的对象状态总是可变的，创建纯粹的不可变对象比较困难，并且使用不可变对象描述状态改变也增加了状态维护的难度；故实际实现时，可酌情取舍，允许灰度，满足需求即可。比如：只允许对象的某些状态不可变。

## 访问对象带来的安全隐患

访问不可变对象时，需要防止不可变对象的状态引用外泄，导致外部环境对状态进行变更（比如：不可变对象包含一个list集合属性，当返回list引用时，虽然声明为final的list引用不可改变，但是list中的元素是可以被改变的）；为了保证安全，此时需要额外进行防御性拷贝操作，比如：通过对状态的深拷贝防止状态的引用外泄。

# 应用案例

## CopyOnWriteArrayList

java.util.concurrent.CopyOnWriteArrayList专门针对遍历操作比添加、删除操作更加频繁的场景设计，既可以提升遍历查找过程中更新集合导致异常的情况出现，又可以避免使用互斥锁带来的性能消耗，适用于”读多写少，小数据量”的场景。

## CopyOnWriteArraySet

java.util.concurrent.CopyOnWriteArraySet使用数组实现的Set集合，与CopyOnWriteArrayList的实现原理类似，适用于”读多写少，小数据量”的场景。

## Redis的RDB快照原理

Redis支持将当前数据的快照存成一个数据文件的持久化机制，即RDB快照。这种方法是非常好理解的，但是一个持续写入的数据库如何生成快照呢？Redis借助了fork命令的copy on write机制（私有内存非共享内存）。在生成快照时，将当前进程fork出一个子进程，然后在子进程中循环所有的数据，将数据写成为RDB文件。

我们可以通过Redis的save指令来配置RDB快照生成的时机，比如你可以配置当10分钟以内有100次写入就生成快照，也可以配置当1小时内有 1000次写入就生成快照，也可以多个规则一起实施。这些规则的定义就在Redis的配置文件中，你也可以通过Redis的CONFIG SET命令在Redis运行时设置规则，不需要重启Redis。

|  |
| --- |
| 在redis中配置：  save 900 1 #当900秒内有一条Keys数据被改变时，生成RDB；  save 300 10 #当300秒内有10条Keys数据被改变时，生成RDB；  save 60 10000 #当60秒内有10000条Keys数据被改变时，生成RDB； |

## Redis的AOF Rewrite原理

AOF日志的全称是append only file，从名字上我们就能看出来，**它是一个追加写入的日志文件。与一般数据库的binlog不同的是，AOF文件是可识别的纯文本**，它的内容就是一个个 的Redis标准命令。当然，并不是发送到Redis的所有命令都要记录到AOF日志里面，只有那些会导致数据发生修改的命令才会追加到AOF文件。

**那么每一条修改数据的命令都生成一条日志，那么AOF文件是不是会很大？**  
        答案是肯定的，AOF文件会越来越大，所以Redis又提供了一个功能，叫做AOF rewrite（使用Redis提供了bgrewriteaof命令就可以）。其功能就是重新生成一份AOF文件，新的AOF文件中一条记录的操作只会有一次，而不像一份老文件那样，可能记录了对同一个值的多次操作。其生成过程和RDB类似，也是fork一个进程，直接遍历数据，写入新的AOF临时文件（这个过程和RDB类似，但是是将数据拆分成一条一条写命令的形式的）。在写入新文件的过程中，所有的写操作日志还是会写到原来老的 AOF文件中，同时还会记录在内存缓冲区中。当重完操作完成后，会将所有缓冲区中的日志一次性写入到临时文件中。然后调用原子性的rename命令用新的 AOF文件取代老的AOF文件。(这样的操作，老的AOF可以恢复内存，如果产生新的AOF，老的就不存在了，可用新的AOF文件恢复内存，这样同时解决了AOF不断增长的问题。)AOF是一个写文件操作，其目的是将操作日志写到磁盘上，所以它也同样会遇到我们上面说的写操作的5个流程。

## Redis的数据存储增量Rehash

|  |
| --- |
|  |

**Why：为什么dict中dictht数组长度定义为2？**

1）背景：哈希表需要定期进行rehash操作，以保证效率：

由于使用哈希表存放数据，当哈希表中数据量达到一定大小后，冲突数据相对较多，会降低数据访问效率，故需要增加哈希表大小（桶的个数），并对数据进行rehash，以保证性能；rehash过程 通常的做法是建立一个额外的hash table，将原来的hash table中的数据在新的数据中进行重新输入，从而生成新的hash表，再改变指针指向；

2）原因：因为rehash会消耗一定性能， 为了防止一次性 rehash 期间 阻塞请求，导致 redis服务能力大幅下降，采用增量式rehash，即通过定时任务，定时分配一定的时间片执行rehash，这种增量式 rehash 需要一个额外的 struct dictht 结构来保存rehash过程的中间数据；

**How：redis中dict如何进行再hash？**

1）dict中存放两个dictht，分别编号0，1;

2）默认情况下优先使用0；

3）当执行增量rehash时，将扩容的哈希表存放到1中；

4）待rehash执行完成后，释放0，将0中指针指向1中指针，清空1；

5） 在调用dictFind的时候，可能需要对两张dict表做查询。唯一的优化判断是，当key在ht[0]不存在且不在rehashing状态时，可以速度返回空；如果在rehashing状态，当在ht[0]没值的时候，还需要在ht[1]里查找； dictAdd的时候，如果状态是rehashing，则把值插入到ht[1]，否则ht[0]；

# 相关模式

## Thread Specific Storage模式

## Serial Thread Confinement模式

# 问题思考

1. **如何确保访问不可变对象时，状态不被外界更改？**
2. **面对读多写少的情况，是不是尽可能将对象实现为纯粹的不可变对象就一定好？**
3. **使用不可变对象需要考虑哪些因素？**