

47-原型模式：如何最快速地clone一个HashMap散列表？

对于创建型模式，前面我们已经讲了单例模式、工厂模式、建造者模式，今天我们来讲最后一个：原型模式。

对于熟悉JavaScript语言的前端程序员来说，原型模式是一种比较常用的开发模式。这是因为，有别于Java、C++等基于类的面向对象编程语言，JavaScript是一种基于原型的面向对象编程语言。即便JavaScript现在也引入了类的概念，但它也只是基于原型的语法糖而已。不过，如果你熟悉的是Java、C++等这些编程语言，那在实际的开发中，就很少用到原型模式了。

今天的讲解跟具体某一语言的语法机制无关，而是通过一个clone散列表的例子带你搞清楚：原型模式的应用场景，以及它的两种实现方式：深拷贝和浅拷贝。虽然原型模式的原理和代码实现非常简单，但今天举的例子还是稍微有点复杂的，你要跟上我的思路，多动脑思考一下。

话不多说，让我们正式开始今天的学习吧！

原型模式的原理与应用

如果对象的创建成本比较大，而同一个类的不同对象之间差别不大（大部分字段都相同），在这种情况下，我们可以利用对已有对象（原型）进行复制（或者叫拷贝）的方式来创建新对象，以达到节省创建时间的目的。这种基于原型来创建对象的方式就叫作**原型设计模式**（Prototype Design Pattern），简称**原型模式**。

那何为“对象的创建成本比较大”？

实际上，创建对象包含的申请内存、给成员变量赋值这一过程，本身并不会花费太多时间，或者说对于大部分业务系统来说，这点时间完全是可以忽略的。应用一个复杂的模式，只得到一点点的性能提升，这就是所谓的过度设计，得不偿失。

但是，如果对象中的数据需要经过复杂的计算才能得到（比如排序、计算哈希值），或者需要从RPC、网络、数据库、文件系统等非常慢速的IO中读取，这种情况下，我们就可以利用原型模式，从其他已有对象中直接拷贝得到，而不用每次在创建新对象的时候，都重复执行这些耗时的操作。

这么说还是比较理论，接下来，我们通过一个例子来解释一下刚刚这段话。

假设数据库中存储了大约10万条“搜索关键词”信息，每条信息包含关键词、关键词被搜索的次数、信息最近被更新的时间等。系统A在启动的时候会加载这份数据到内存中，用于处理某些其他的业务需求。为了方便快速地查找某个关键词对应的信息，我们给关键词建立一个散列表索引。

如果你熟悉的是Java语言，可以直接使用语言中提供的HashMap容器来实现。其中，HashMap的key为搜索关键词，value为关键词详细信息（比如搜索次数）。我们只需要将数据从数据库中读取出来，放入HashMap就可以了。

不过，我们还有另外一个系统B，专门用来分析搜索日志，定期（比如间隔10分钟）批量地更新数据库中的数据，并且标记为新的数据版本。比如，在下面的示例图中，我们对v2版本的数据进行更新，得到v3版本的数据。这里我们假设只有更新和新添关键词，没有删除关键词的行为。

v2			v3		
关键词	次数	更新时间戳	关键词	次数	更新时间戳
算法	2098	1548506764	算法	2098	1548506764
设计模式	1938	1548470987	设计模式	2188	1548513456（更新）
小争哥	13098	1548384124	小争哥	13098	1548384124
.....	王争	234	1548513781（新添）
.....



为了保证系统A中数据的实时性（不一定非常实时，但数据也不能太旧），系统A需要定期根据数据库中的数据，更新内存中的索引和数据。

我们该如何实现这个需求呢？

实际上，也不难。我们只需要在系统A中，记录当前数据的版本Va对应的更新时间Ta，从数据库中捞出更新时间大于Ta的所有搜索关键词，也就是找出Va版本与最新版本数据的“差集”，然后针对差集中的每个关键词进行处理。如果它已经在散列表中存在了，我们就更新相应的搜索次数、更新时间等信息；如果它在散列表中不存在，我们就将它插入到散列表中。

按照这个设计思路，我给出的示例代码如下所示：

```
public class Demo {
    private ConcurrentHashMap<String, SearchWord> currentKeywords = new ConcurrentHashMap<>();
    private long lastUpdateTime = -1;

    public void refresh() {
        // 从数据库中取出更新时间>lastUpdateTime的数据，放入到currentKeywords中
        List<SearchWord> toBeUpdatedSearchWords = getSearchWords(lastUpdateTime);
        long maxNewUpdatedTime = lastUpdateTime;
        for (SearchWord searchWord : toBeUpdatedSearchWords) {
            if (searchWord.getLastUpdateTime() > maxNewUpdatedTime) {
                maxNewUpdatedTime = searchWord.getLastUpdateTime();
            }
            if (currentKeywords.containsKey(searchWord.getKeyword())) {
                currentKeywords.replace(searchWord.getKeyword(), searchWord);
            } else {
                currentKeywords.put(searchWord.getKeyword(), searchWord);
            }
        }

        lastUpdateTime = maxNewUpdatedTime;
    }

    private List<SearchWord> getSearchWords(long lastUpdateTime) {
        // TODO: 从数据库中取出更新时间>lastUpdateTime的数据
        return null;
    }
}
```

```
}
```

不过，现在，我们有一个特殊的要求：任何时刻，系统A中的所有数据都必须是同一个版本的，要么都是版本a，要么都是版本b，不能有的是版本a，有的是版本b。那刚刚的更新方式就不能满足这个要求了。除此之外，我们还要求：在更新内存数据的时候，系统A不能处于不可用状态，也就是不能停机更新数据。

那我们该如何实现现在这个需求呢？

实际上，也不难。我们把正在使用的数据的版本定义为“服务版本”，当我们要更新内存中的数据的时候，我们并不是直接在服务版本（假设是版本a数据）上更新，而是重新创建另一个版本数据（假设是版本b数据），等新的版本数据建好之后，再一次性地将服务版本从版本a切换到版本b。这样既保证了数据一直可用，又避免了中间状态的存在。

按照这个设计思路，我给出的示例代码如下所示：

```
public class Demo {
    private HashMap<String, SearchWord> currentKeywords=new HashMap<>();

    public void refresh() {
        HashMap<String, SearchWord> newKeywords = new LinkedHashMap<>();

        // 从数据库中取出所有的数据，放入到newKeywords中
        List<SearchWord> toBeUpdatedSearchWords = getSearchWords();
        for (SearchWord searchWord : toBeUpdatedSearchWords) {
            newKeywords.put(searchWord.getKeyword(), searchWord);
        }

        currentKeywords = newKeywords;
    }

    private List<SearchWord> getSearchWords() {
        // TODO: 从数据库中取出所有的数据
        return null;
    }
}
```

不过，在上面的代码实现中，newKeywords构建的成本比较高。我们需要将这10万条数据从数据库中读出，然后计算哈希值，构建newKeywords。这个过程显然是比较耗时。为了提高效率，原型模式就派上用场了。

我们拷贝currentKeywords数据到newKeywords中，然后从数据库中只捞出新增或者有更新的关键词，更新到newKeywords中。而相对于10万条数据来说，每次新增或者更新的关键词个数是比较少的，所以，这种策略大大提高了数据更新的效率。

按照这个设计思路，我给出的示例代码如下所示：

```
public class Demo {
```

```

private HashMap<String, SearchWord> currentKeywords=new HashMap<>();
private long lastUpdateTime = -1;

public void refresh() {
    // 原型模式就这么简单，拷贝已有对象的数据，更新少量差值
    HashMap<String, SearchWord> newKeywords = (HashMap<String, SearchWord>) currentKeywords.clone();

    // 从数据库中取出更新时间>lastUpdateTime的数据，放入到newKeywords中
    List<SearchWord> toBeUpdatedSearchWords = getSearchWords(lastUpdateTime);
    long maxNewUpdatedTime = lastUpdateTime;
    for (SearchWord searchWord : toBeUpdatedSearchWords) {
        if (searchWord.getLastUpdateTime() > maxNewUpdatedTime) {
            maxNewUpdatedTime = searchWord.getLastUpdateTime();
        }
        if (newKeywords.containsKey(searchWord.getKeyword())) {
            SearchWord oldSearchWord = newKeywords.get(searchWord.getKeyword());
            oldSearchWord.setCount(searchWord.getCount());
            oldSearchWord.setLastUpdateTime(searchWord.getLastUpdateTime());
        } else {
            newKeywords.put(searchWord.getKeyword(), searchWord);
        }
    }

    lastUpdateTime = maxNewUpdatedTime;
    currentKeywords = newKeywords;
}

private List<SearchWord> getSearchWords(long lastUpdateTime) {
    // TODO: 从数据库中取出更新时间>lastUpdateTime的数据
    return null;
}
}

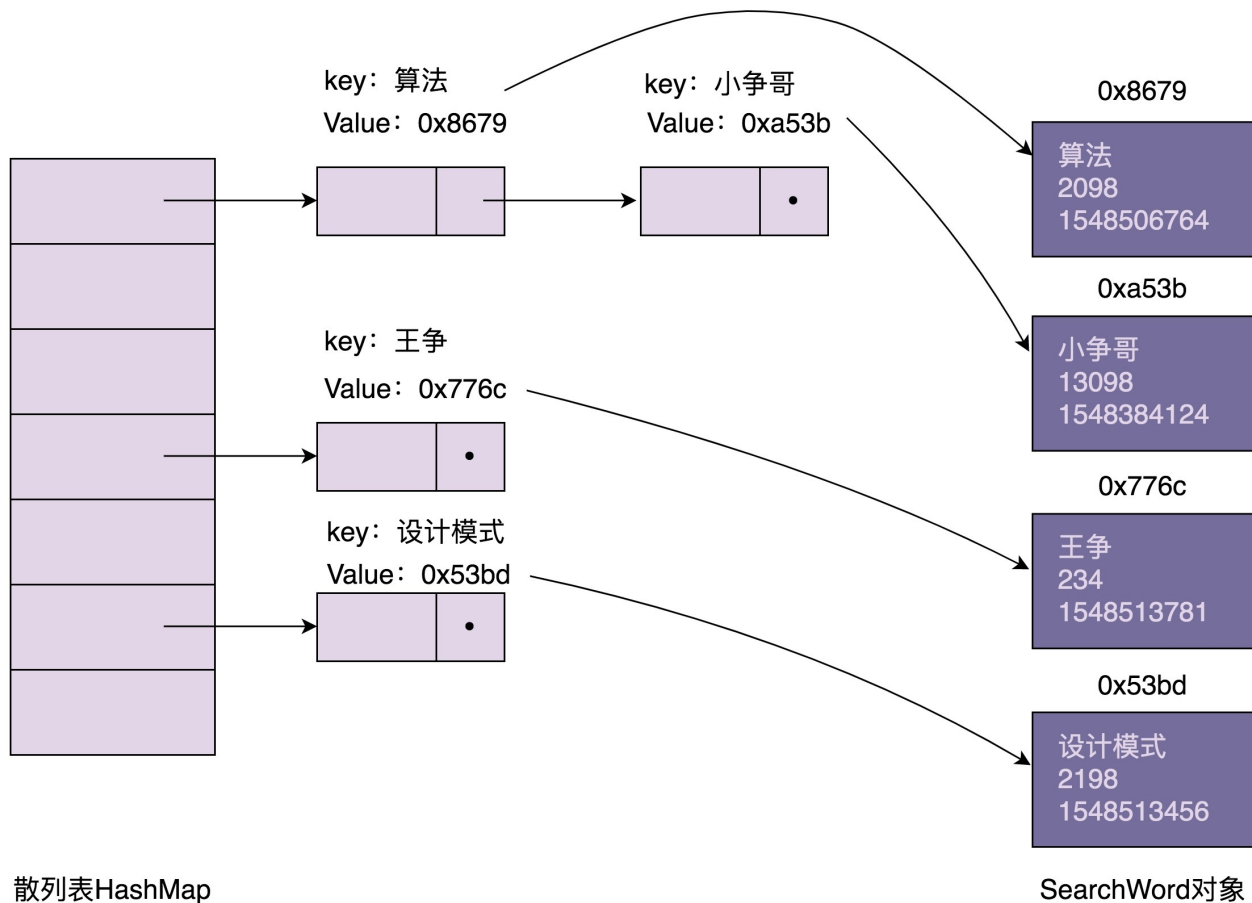
```

这里我们利用了Java中的clone()语法来复制一个对象。如果你熟悉的语言没有这个语法，那把数据从currentKeywords中一个个取出来，然后再重新计算哈希值，放入到newKeywords中也是可以接受的。毕竟，最耗时的还是从数据库中取数据的操作。相对于数据库的IO操作来说，内存操作和CPU计算的耗时都是可以忽略的。

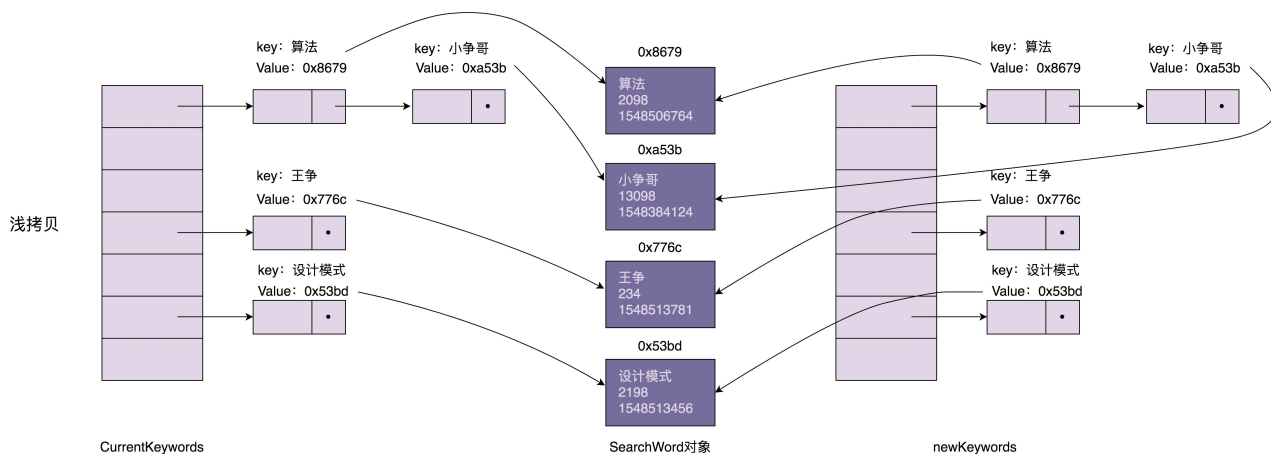
不过，不知道你有没有发现，实际上，刚刚的代码实现是有问题的。要弄明白到底有什么问题，我们需要先了解另外两个概念：深拷贝（Deep Copy）和浅拷贝（Shallow Copy）。

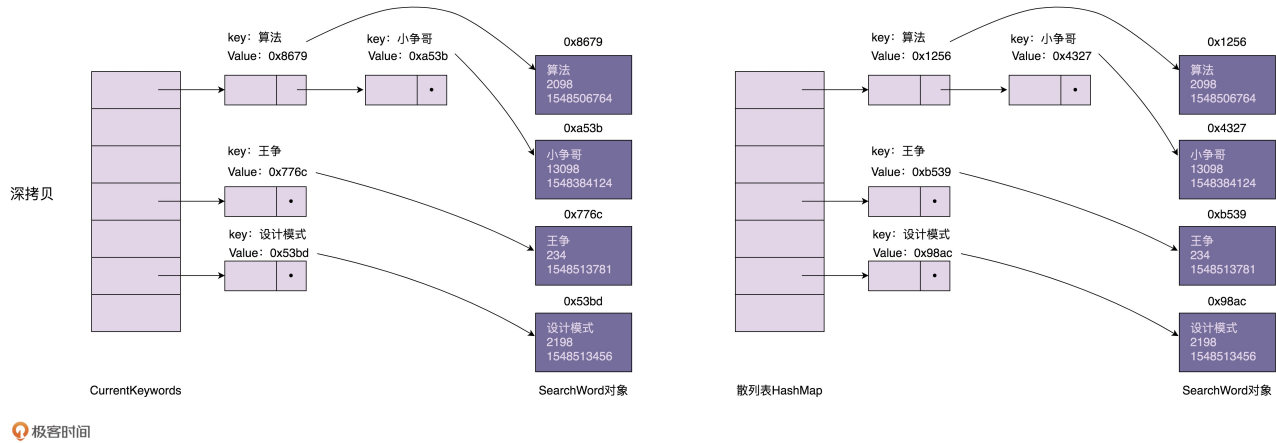
原型模式的实现方式：深拷贝和浅拷贝

我们来看，在内存中，用散列表组织的搜索关键词信息是如何存储的。我画了一张示意图，大致结构如下所示。从图中我们可以发现，散列表索引中，每个结点存储的key是搜索关键词，value是SearchWord对象的内存地址。SearchWord对象本身存储在散列表之外的内存空间中。



浅拷贝和深拷贝的区别在于，浅拷贝只会复制图中的索引（散列表），不会复制数据（SearchWord对象）本身。相反，深拷贝不仅仅会复制索引，还会复制数据本身。浅拷贝得到的对象（newKeywords）跟原始对象（currentKeywords）共享数据（SearchWord对象），而深拷贝得到的是一份完完全全独立的对象。具体的对比如下图所示：





在Java语言中，Object类的clone()方法执行的就是我们刚刚说的浅拷贝。它只会拷贝对象中的基本数据类型的数据（比如，int、long），以及引用对象（SearchWord）的内存地址，不会递归地拷贝引用对象本身。

在上面的代码中，我们通过调用HashMap上的clone()浅拷贝方法来实现原型模式。当我们通过newKeywords更新SearchWord对象的时候（比如，更新“设计模式”这个搜索关键词的访问次数），newKeywords和currentKeywords因为指向相同的一组SearchWord对象，就会导致currentKeywords中指向的SearchWord，有的是老版本的，有的是新版本的，就没法满足我们之前的需求：currentKeywords中的数据在任何时刻都是同一个版本的，不存在介于老版本与新版本之间的中间状态。

现在，我们又该如何来解决这个问题呢？

我们可以将浅拷贝替换为深拷贝。newKeywords不仅仅复制currentKeywords的索引，还把SearchWord对象也复制一份出来，这样newKeywords和currentKeywords就指向不同的SearchWord对象，也就不存在更新newKeywords的数据会导致currentKeywords的数据也被更新的问题了。

那如何实现深拷贝呢？总结一下的话，有下面两种方法。

第一种方法：递归拷贝对象、对象的引用对象以及引用对象的引用对象……直到要拷贝的对象只包含基本数据类型数据，没有引用对象为止。根据这个思路对之前的代码进行重构。重构之后的代码如下所示：

```
public class Demo {
    private HashMap<String, SearchWord> currentKeywords=new HashMap<>();
    private long lastUpdateTime = -1;

    public void refresh() {
        // Deep copy
        HashMap<String, SearchWord> newKeywords = new HashMap<>();
        for (HashMap.Entry<String, SearchWord> e : currentKeywords.entrySet()) {
            SearchWord searchWord = e.getValue();
            SearchWord newSearchWord = new SearchWord(
                searchWord.getKeyword(), searchWord.getCount(), searchWord.getLastUpdateTime());
            newKeywords.put(e.getKey(), newSearchWord);
        }

        // 从数据库中取出更新时间>lastUpdateTime的数据，放入到newKeywords中
        List<SearchWord> toBeUpdatedSearchWords = getSearchWords(lastUpdateTime);
        long maxNewUpdatedTime = lastUpdateTime;
        for (SearchWord searchWord : toBeUpdatedSearchWords) {
            if (searchWord.getLastUpdateTime() > maxNewUpdatedTime) {
```

```

        maxNewUpdatedTime = searchWord.getLastUpdateTime();
    }
    if (newKeywords.containsKey(searchWord.getKeyword())) {
        SearchWord oldSearchWord = newKeywords.get(searchWord.getKeyword());
        oldSearchWord.setCount(searchWord.getCount());
        oldSearchWord.setLastUpdateTime(searchWord.getLastUpdateTime());
    } else {
        newKeywords.put(searchWord.getKeyword(), searchWord);
    }
}

lastUpdateTime = maxNewUpdatedTime;
currentKeywords = newKeywords;
}

private List<SearchWord> getSearchWords(long lastUpdateTime) {
    // TODO: 从数据库中取出更新时间>lastUpdateTime的数据
    return null;
}
}

```

第二种方法：先将对象序列化，然后再反序列化成新的对象。具体的示例代码如下所示：

```

public Object deepCopy(Object object) {
    ByteArrayOutputStream bo = new ByteArrayOutputStream();
    ObjectOutputStream oo = new ObjectOutputStream(bo);
    oo.writeObject(object);

    ByteArrayInputStream bi = new ByteArrayInputStream(bo.toByteArray());
    ObjectInputStream oi = new ObjectInputStream(bi);

    return oi.readObject();
}

```

刚刚的两种实现方法，不管采用哪种，深拷贝都要比浅拷贝耗时、耗内存空间。针对我们这个应用场景，有没有更快、更省内存的实现方式呢？

我们可以先采用浅拷贝的方式创建newKeywords。对于需要更新的SearchWord对象，我们再使用深度拷贝的方式创建一份新的对象，替换newKeywords中的老对象。毕竟需要更新的数据是很少的。这种方式即利用了浅拷贝节省时间、空间的优点，又能保证currentKeywords中的中数据都是老版本的数据。具体的代码实现如下所示。这也是标题中讲到的，在我们这个应用场景下，最快速clone散列表的方式。

```

public class Demo {
    private HashMap<String, SearchWord> currentKeywords=new HashMap<>();
    private long lastUpdateTime = -1;

    public void refresh() {
        // Shallow copy
        HashMap<String, SearchWord> newKeywords = (HashMap<String, SearchWord>) currentKeywords.clone();

        // 从数据库中取出更新时间>lastUpdateTime的数据，放入到newKeywords中
    }
}

```



```
List<SearchWord> toBeUpdatedSearchWords = getSearchWords(lastUpdateTime);
long maxNewUpdatedTime = lastUpdateTime;
for (SearchWord searchWord : toBeUpdatedSearchWords) {
    if (searchWord.getLastUpdateTime() > maxNewUpdatedTime) {
        maxNewUpdatedTime = searchWord.getLastUpdateTime();
    }
    if (newKeywords.containsKey(searchWord.getKeyword())) {
        newKeywords.remove(searchWord.getKeyword());
    }
    newKeywords.put(searchWord.getKeyword(), searchWord);
}

lastUpdateTime = maxNewUpdatedTime;
currentKeywords = newKeywords;
}

private List<SearchWord> getSearchWords(long lastUpdateTime) {
    // TODO: 从数据库中取出更新时间>lastUpdateTime的数据
    return null;
}
}
```

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

1.什么是原型模式？

如果对象的创建成本比较大，而同一个类的不同对象之间差别不大（大部分字段都相同），在这种情况下，我们可以利用对已有对象（原型）进行复制（或者叫拷贝）的方式，来创建新对象，以达到节省创建时间的目的。这种基于原型来创建对象的方式就叫作原型设计模式，简称原型模式。

2.原型模式的两种实现方法

原型模式有两种实现方法，深拷贝和浅拷贝。浅拷贝只会复制对象中基本数据类型数据和引用对象的内存地址，不会递归地复制引用对象，以及引用对象的引用对象……而深拷贝得到的是一份完完全全独立的对象。所以，深拷贝比起浅拷贝来说，更加耗时，更加耗内存空间。

如果要拷贝的对象是不可变对象，浅拷贝共享不可变对象是没问题的，但对于可变对象来说，浅拷贝得到的对象和原始对象会共享部分数据，就有可能出现数据被修改的风险，也就变得复杂多了。除非像我们今天实战中举的那个例子，需要从数据库中加载10万条数据并构建散列表索引，操作非常耗时，比较推荐使用浅拷贝，否则，没有充分的理由，不要为了一点点的性能提升而使用浅拷贝。

课堂讨论

1. 在今天的应用场景中，如果不仅往数据库中添加和更新关键词，还删除关键词，这种情况下，又该如何实现呢？
2. 在[第7讲](#)中，为了让ShoppingCart的getItems()方法返回不可变对象，我们如下来实现代码。当时，我们指出这样的实现思路还是有点问题。因为当调用者通过ShoppingCart的getItems()获取到items之后，我们还是可以修改容器中每个对象（ShoppingCartItem）的数据。学完本节课之后，现在你有没有解决方法了呢？


```
public class ShoppingCart {
    // ...省略其他代码...
    public List<ShoppingCartItem> getItems() {
        return Collections.unmodifiableList(this.items);
    }
}

// Testing Code in main method:
ShoppingCart cart = new ShoppingCart();
List<ShoppingCartItem> items = cart.getItems();
items.clear();//try to modify the list
// Exception in thread "main" java.lang.UnsupportedOperationException

ShoppingCart cart = new ShoppingCart();
cart.add(new ShoppingCartItem(...));
List<ShoppingCartItem> items = cart.getItems();
ShoppingCartItem item = items.get(0);
item.setPrice(19.0); // 这里修改了item的价格属性
```

欢迎留言和我分享你的疑惑和见解，如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言：

- 平风造雨 2020-02-19 23:36:14

1. 两个Map比较下key找到差集

2. 可以返回深复制的购物车结构，或者干脆分成两个方法，一个返回深复制的结构，一个返回当前结构，区分使用场景。 [1赞]

- 辣么大 2020-02-19 20:58:03

问题1:

方法一：新旧的数据取交集，可以删除旧map中的删除关键字，之后的逻辑就和文章中一样了。

方法二：逻辑删除，当map的size中已删除占比过高时，resize map。

争哥说：这里我们利用了 Java 中的 clone() 语法来复制一个对象。如果你熟悉的语言没有这个语法，那把数据从 currentKeywords 中一个个取出来，然后再重新计算哈希值，放入到 newKeywords 中也是可以接受的。

Java HashMap的clone方法就把数据取出来，计算hash值，在放回去的。clone方法中，调用了putMap Entries方法，其中有一关键的一行，克隆重新计算了hash值：
putVal(hash(key), key, value, false, evict);

文章中的深复制：为什么SearchWord不重写clone方法呢？

@Override

```
protected Object clone() throws CloneNotSupportedException {
    SearchWord newWord = new SearchWord(this.keyWord, this.times, this.timestamp);
    return newWord;
}
```

[1赞]

- Frank 2020-02-19 22:40:04

原型模式是一种从“拷贝”的角度来创建对象的方式，以实现节省时间的目的。原型模式有两种实现方式

：浅拷贝与深拷贝。理解其应用场景：对象创建成本大，同一个类不同对象之间差别不大。

- 我来也 2020-02-19 22:37:14

思考题2:

即使是深拷贝，也是可以修改的，只是修改的不是原数据而已。

我对java语法不熟，不知道可否递归的使用 Collections.unmodifiableList 类似的方式，构建一个新的深拷贝对象。然后再返回这个对象的不可修改副本。

这样这个对象从里到外都是不可修改的属性了。

- 忆水寒 2020-02-19 21:28:40

让我想到了linux下面fork，其实内核也是拷贝了一份数据。Java里面的copyonwrite是不是也是这种深拷贝原理呢？

- L🐼 2020-02-19 15:51:17

问题 1: 逻辑删除即可

问题 2: 返回深拷贝对象

- aoe 2020-02-19 15:04:43

原来拷贝可以这样操作！佩服小争哥算法与数据结构用的6！

- , 2020-02-19 13:28:56

问题1:

1.设置标记位,"使用中","弃用","已删除"等,检查到标记为"弃用"的数据时,删除map里的数据同时修改标记位为"已删除",扫描数据库里更新的数据时增加检索条件"使用中"

2.每次数据库都全量扫描,拿到标记位为"使用中"的数据,直接替换map

3.删除数据时同时删除数据库和map里的数据

个人认为方法1和方法3都很合适,方法二对数据库IO操作量比较大,不太适合

问题2:

返回深克隆对象即可

- Jxin 2020-02-19 13:06:53

1.逻辑删除的话，代码都不用改。物理删除的话，我觉得在删除时联动清除map的缓存可行（单进程，分布式就得引入一个外部存储，告知所有节点删除某个缓存）。

2.根据业务场景，采用cow写时复制。提供只读的列表返回和写时的复制列表的返回两个方法。

- webmin 2020-02-19 11:37:55

问题一：

```
for (SearchWord searchWord : toBeUpdatedSearchWords) {
```

```
...
```

```
}
```

```
Set<String> toBeUpdatedKeys = new HashSet<>();
```

```
toBeUpdatedSearchWords.forEach((k) -> toBeUpdatedKeys.add(k.getKeyword()));
```

```
List<String> removeList = newKeywords.keySet().stream().filter((key) -> !toBeUpdatedKeys.contains(key)).collect(Collectors.toList());
```

```
removeList.forEach(newKeywords::remove);
```

问题二：

返回不可变对象，方法有两个

方法1：语言自带或第三方库的不可变对象机制，如：Java可以使用第三方库Guava的不可变集合；

方法2：返回一个深拷贝对象；

- 小晏子 2020-02-19 10:59:54

1. 考虑到删除关键词，那么最好数据库使用软删除，这样可以知道哪些关键词是被删除的，那么拿到这些被删除的关键词就可以在clone出来的newKeywords基础上，直接remove掉已经删除的哪些关键词就可以了。反之如果不是使用的软删除，那么就不好使用原型模式，需要获取新版本全量数据，然后和旧版本数据一一比对，看哪些数据是被删除的了。

2. 代码如下，将原来的items deep clone一份，这样就切断了与原来items的联系。

```
public class ShoppingCart {  
    // ...省略其他代码...  
    public List<ShoppingCartItem> getItems() {  
        List<ShoppingCartItem> tmpShoppingCartItems = new ArrayList<>();  
        tmpShoppingCartItems.addAll(this.items);  
        return Collections.unmodifiableList(tmpShoppingCartItems);  
    }  
}
```

- 唐龙 2020-02-19 10:43:31

之前听说，可能你在不经意间已经用过一些设计模式了，今天终于有这种感觉了，确实对原型模式有过一些简单应用。

- test 2020-02-19 09:29:17

1. 用一个标记位表示被删除
2. 返回一个深拷贝对象

- Panmax 2020-02-19 09:21:56

文章中的最后一种方案：「先采用浅拷贝的方式创建 newKeywords。对于需要更新的 SearchWord 对象，再使用深度拷贝的方式创建一份新的对象，替换 newKeywords 中的老对象。」

是不是可以理解为是一种 Copy On Write 的优化策略

- pedro 2020-02-19 09:21:12

想到了linux进程的克隆，c++的拷贝构造函数都是一种原型模式的使用

- Summer 空城 2020-02-19 08:39:14

- 1, 删除key对于clone的对象而言，不会影响之前的对象，所以实现应该不需要变化吧
- 2, return new ArrayList<>(this.items);

- Jeff.Smile 2020-02-19 00:10:25

沙发