Теория и практика Java: Изменять или не изменять?

Неизменяемые объекты могут значительно облегчить вашу жизнь

Брайан Гетц
главный консультант
Quiotix

08.02.2007

Неизменяемые объекты имеют несколько свойств, которые облегчают работу с ними, включая более мягкие требования к синхронизации и возможность разделять и кэшировать ссылки на объекты без опасения повреждения данных. Неизменяемость не обязательно имеет смысл для всех классов, однако большинство программ имеют по меньшей мере несколько классов, для которых неизменяемость была бы полезна. В выпуске *Теории и практики Java* за этот месяц Брайан Гетц объясняет некоторые из преимуществ неизменяемости и дает некоторые рекомендации по построению неизменяемых классов.

Больше статей из этой серии

Неизменяемый объект - это такой объект, чье внешнее видимое состояние не может измениться после его создания. Классы `String`, `Integer` и `BigDecimal` в библиотеке классов Java являются примерами неизменяемых объектов - они представляют отдельное значение, которое не может измениться в течение жизненного цикла объекта.

Преимущества неизменяемости

Неизменяемые классы при правильном использовании могут значительно упростить программирование. Они могут находиться только в одном состоянии, поэтому если они правильно сконструированы, они никак не могут быть в несогласованном состоянии. Вы можете свободно делать общими и кэшировать ссылки на неизменяемые объекты без необходимости копировать или клонировать их; вы можете кэшировать их поля или результаты их методов, не беспокоясь о том, что значения устареют или станут несогласованными с остальными состояниями объекта. Из неизменяемых классов обычно получаются лучшие map key. По существу они поточно-ориентированы, поэтому вам не нужно синхронизировать доступ к ним через потоки.

Возможность кэширования

Поскольку нет опасности, что неизменяемые объекты изменят свое значение, можно свободно кэшировать ссылки на них и быть уверенным, что эта ссылка обратится к этому же значению позже. Соответственно, поскольку их свойства не могут измениться, можно кэшировать их поля и результаты их методов.

Если объект изменяемый, вам нужно быть осторожными, сохраняя ссылку на него. Рассмотрим код в Листинге 1, где в очередь на выполнение планировщиком поставлены два задания . Цель в том, чтобы первая задача начала выполняться сейчас, а вторая - через один день.

Листинг 1. Потенциальная проблема с изменяемым объектом Date (дата)

```
Date d = new Date();
Scheduler.scheduleTask(task1, d);
d.setTime(d.getTime() + ONE_DAY);
scheduler.scheduleTask(task2, d);
```

Поскольку `Date` - изменяемый объект, метод `scheduleTask` должен на всякий случай скопировать параметр даты (возможно через `clone()`) во внутреннюю структуру данных. Иначе как `task1`, так и `task2` могут выполниться завтра, что не соответствует нашей цели. Хуже того, внутренняя структура данных, использующаяся планировщиком задач, может быть повреждена. Очень легко забыть скопировать параметр даты при написании метода типа `scheduleTask()`. Если вы об этом все же забудете, то создадите неуловимую ошибку, которая некоторое время не будет проявляться, а когда проявится, потребуется долгое время, чтобы ее отследить. Неизменяемый класс `Date` сделал бы ошибку такого рода невозможной.

Внутренняя безопасность потоков

Многие вопросы, связанные с безопасностью потоков, возникают, когда множественные потоки пытаются параллельно модифицировать состояние объекта (конфликт по совпадению обращений для записи) или когда один поток пытается получить доступ к состоянию объекта, в то время как другой поток модифицирует его (конфликт по совпадению обращений при считывании и записи.) Для предупреждения таких конфликтов необходимо синхронизировать доступ к совместно используемым объектам, так, чтобы другие потоки не могли получить к ним доступ, пока они в несогласованном состоянии. Возможно, это будет трудно осуществить правильно, требуется значительное количество документации для того, чтобы программа была правильно расширена, а также могут возникнуть отрицательные последствия, связанные с производительностью. Если неизменяемые объекты сконструированы правильно (это означает, что ссылка на объект покинула пределы конструктора), нет необходимости синхронизировать доступ, так как их состояние не может быть изменено, и поэтому конфликт по совпадению обращений для записи или по совпадению обращений при считывании и записи не может возникнуть.

Возможность делать общими ссылки на неизменяемые объекты в потоках без синхронизации может значительно облегчить процесс синхронизации написания параллельных программ и снижает количество потенциальных ошибок в них.

Безопасность при работе с "плохим" кодом

Методы, которые принимают объекты в качестве аргументов, не должны изменять состояние этих объектов, кроме тех случаев, когда им это документально предписано или они становятся владельцами этого объекта. Когда мы передаем объект обычному методу, мы обычно не ожидаем его возвращения в неизмененном виде. Однако по отношению к изменяемым объектам такая уверенность просто неразумна. Если мы передаем `java.awt.Point` такому методу, как `Component.setLocation()`, ничто не мешает `setLocation` модифицировать местоположение передаваемого `Point` или сохранить ссылку на этот point и изменить его позже в другом методе. (Конечно, `Component` этого не делает, потому, что это было бы "грубо", но не все классы такие "вежливые".) Теперь состояние нашего объекта `Point` изменилось без нашего ведома, и результаты потенциально опасны - мы все еще думаем, что point в одном месте, а он, фактически, находится в другом. Однако, если бы объект`Point` был неизменяемым, такой "враждебный" код не смог бы модифицировать состояние нашей программы таким сбивающим с толку и опасным способом.

Хорошие ключи

Из неизменяемых объектов получаются лучшие ключи `HashMap` или `HashSet`. Некоторые изменяемые объекты изменяют значение `hashcode()` в зависимости от своего состояния (как, например, класс `StringHolder` в Листинге 2). Если вы используете такой изменяемый объект, как ключ `HashSet`, а затем объект изменяет свое состояние, при реализации `HashSet` может возникнуть путаница - объект все еще будет присутствовать, если пронумеровать набор (set), но может оказаться, что его нет, если обратиться с запросом к набору при помощи `contains()`. Нет необходимости говорить, что это могло бы вызвать непредсказуемое поведение. Код Листинга 2, где это демонстрируется, напечатает "false", "1," и "moo."

Листинг 2. Изменяемый класс StringHolder, неподходящий для использования в качестве ключа

```
    public class StringHolder  {
        private String string

        public StringHolder(string's) {
            this.string = s;
        }

        public string getstring() {
            return string;
        }

        public void setstring(string string) {
            this.string = string;
        }

        public boolean equals(Object o) {
            if (this == o)
                return true;
            else if (o == null || !(o instanceof stringHolder))
                return false;
            else {
                final stringHolder other = (stringHolder) o;
                if (string == null)
                    return (other.string == null);
                else
                    return string);
```

```
            }
        }

        public int hashcode() {
            return (string != null ? string.hashcode() : 0);
        }

        public string tostring() {
            return string;
        }

        ...

        stringHolder sh = new stringHolder(";blert";);
        HashSet h = new HashSet();
        h.add(sh);
        sh.setstring(";moo";);
        System.out.println(h.contains(sh));
        System.out.println(h.size());
        System.out.println(h.iterator().next());
    }
```

Когда использовать неизменяемые классы

Неизменяемые классы идеальны для представления значения абстрактных типов данных, таких как числа, перечислимые типы или цвета. Основные числовые классы в библиотеке классов Java, такие как `Integer`, `Long` и `Float` - неизменяемые, так же как и стандартные числовые типы, такие как `BigInteger` и `BigDecimal`. Классам для представления сложных чисел или рациональных чисел произвольной точности лучше обладать неизменяемостью. В зависимости от вашего приложения даже абстрактные типы, которые содержат много дискретных значений, таких как векторы или матрицы, могли бы применяться как неизменяемые классы.

Шаблон Flyweight

Неизменяемость - это то, что позволяет шаблону Flyweight, который использует совместный доступ для облегчения использования объектов, фактически представлять большое количество мелкоструктурных объектов. Например, вам может понадобиться представить каждый символ текстового документа или каждый пиксел изображения при помощи объекта, но "наивная" реализация такой стратегии потребовала бы чрезмерно больших затрат памяти и могло бы вызвать проблемы с управлением ею. Шаблон Flyweight применяет фабричный метод (factory method) для распределения ссылок на неизменяемые мелкоструктурные объекты, а также задействует совместное использование, чтобы вести счет объектов, имея лишь один экземпляр объекта, соответствующий букве "a." Дополнительную информацию по шаблону Flyweight можно найти в классической книге *Конструктивные шаблоны* (Gamma и др.; см. Ресурсы).

Еще один пример неизменяемости в библиотеке классов Java - `java.awt.Color`. Поскольку цвета обычно представляются в виде упорядоченного набора числовых значений в некоторых способах цветового представления (таких как RGB, HSB или CMYK), кажется более разумным рассматривать цвет как выделенное значение в пространстве цветов, а не как упорядоченный набор индивидуально адресуемых значений, и поэтому есть смысл реализовать `Color` как неизменяемый класс.

Следует ли представлять объекты, являющиеся контейнерами для множественных элементарных значений, таких как точки, векторы, матрицы или цвета RGB, при помощи изменяемых или неизменяемых объектов? Ответ - когда как... Как они будут использованы?

Используются ли они главным образом для представления многомерных значений (например, цвета пиксела), или просто как контейнеры для совокупности связанных друг с другом свойств некоторого другого объекта (например, высоты и ширины окна)? Как часто эти свойства будут изменяться? Если они изменяются, имеют ли значения индивидуального компонента собственный смысл в приложении?

События - еще один яркий пример возможной реализации неизменяемых классов. События кратковременны, и часто используются не в том потоке, где были созданы, поэтому использование их как неизменяемых имеет больше преимуществ, чем недостатков. Большая часть классов событий AWT реализуются не как строго неизменяемые, а с небольшими модификациями. Соответственно, в системе, которая использует какую-либо форму обмена сообщениями для общения между компонентами, использование пересылаемых объектов в качестве неизменяемых, вероятно, является разумным.

Рекомендации по написанию неизменяемых классов

Написать неизменяемые классы несложно. Класс будет неизменяемым, если верно следующее:

- Все его поля являются final
- Класс объявляется как final
- Ссылка `this` не должна пропасть во время конструирования
- Любые поля, содержащие ссылки на изменяемые объекты, например массивы, совокупности или изменяемые классы, например `Date`:
  - Являются приватными
  - Никогда не возвращаются и никаким другим образом не становятся доступными вызывающим операторам
  - Являются единственными ссылками на те объекты, на которые они ссылаются
  - Не изменяют после конструирования состояние объектов, на которые они ссылаются

Последняя группа требований кажется сложной, но это главным образом означает, что если вы собираетесь хранить ссылку на массив или другие изменяемые объекты, вы должны обеспечить своему классу монопольный доступ к этому изменяемому объекту (так как в ином случае кто-то другой сможет изменить его состояние) и убедиться, что вы не модифицировали его состояние после конструирования. Это усложнение необходимо, чтобы позволить неизменяемым объектам хранить ссылки на массивы, поскольку язык Java не имеет возможности обеспечить условия, когда элементы результирующих массивов не будут модифицироваться. Помните, что если ссылки на массив или другие изменяемые поля инициализируются аргументами, переданных конструктору, вам необходимо не всякий случай скопировать аргументы вызывающей стороны, иначе вы не можете быть уверены, что у вас монопольный доступ к массиву. В ином случае, вызывающая сторона сможет модифицировать состояние массива после вызова конструктора. Листинг 3 показывает правильный и неправильный способы написания конструктора для неизменяемого объекта, который хранит массив вызывающей стороны.

Листинг 3. Правильный и неправильный способы кодирования неизменяемых объектов

```
class ImmutableArrayHolder {

  private final int[] theArray;

  // Right way to write a constructor -- copy the array
  public ImmutableArrayHolder(int[] anArray) {
    this.theArray = (int[]) anArray.clone();
  }

  // Wrong way to write a constructor -- copy the reference
  // The caller could change the array after the call to the constructor
  public ImmutableArrayHolder(int[] anArray) {
    this.theArray = anArray;
  }

  // Right way to write an accessor -- don't expose the array reference
  public int getArrayLength() { return theArray.length }
  public int getArray(int n)  { return theArray[n]; }

  // Right way to write an accessor -- use clone()
  public int[] getArray()       { return (int[]) theArray.clone(); }

  // Wrong way to write an accessor -- expose the array reference
  // A caller could get the array reference and then change the contents
  public int[] getArray()       { return theArray }
}
```

Выполнив некоторые дополнительные действия, можно создать неизменяемые классы, которые используют некоторые неокончательные (non-final) поля (например, стандартная реализация `String` предполагает использование отложенного вычисления (lazy computation) значения `hashCode`), которое может работать лучше, чем строго окончательные классы. Если ваш класс представляет значение абстрактного типа, как, например, числовой тип или цвет, вам также понадобится реализовать методы `hashcode()` и `equals()`, для того, чтобы ваш объект работал хорошо в качестве ключа в `HashMap` или `HashSet`. Для поддержания безопасности потоков важно, чтобы ссылка `this` не выходила из конструктора.

Редко меняющиеся данные

Некоторые элементы данных остаются неизменяемыми все время действия программы, в то время как другие часто изменяются. Неизменяемые данные - очевидные претенденты на неизменяемость, а объекты со сложными и часто меняющимися состояниями обычно не подходят для реализации с неизменяемыми классами. А как насчет данных, которые изменяются иногда, но не часто? Есть ли способ достичь удобства и безопасности потоков, преимуществ, связанных с неизменяемостью, для данных, которые *иногда* изменяются?

Класс `copyOnWritearrayList` из пакета `util.concurrent` - яркий пример того, как можно использовать силу неизменяемости, в то же время иногда допуская модификации. Он идеален для использования классами, которые поддерживают приемники событий, такие как компоненты пользовательского интерфейса. Список приемников событий может измениться, однако он изменяется гораздо реже, чем генерируются события.

`copyOnWritearrayList` ведет себя во многом как класс `arrayList`, за исключением того, что, когда список модифицируется, вместо изменения базового массива создается новый массив, а старый отбрасывается. Это значит, что когда вызывающая сторона получает итератор, который изнутри содержит ссылку на базовый массив, массив, на который итератор ссылается, фактически неизменяем и поэтому можно обойтись без синхронизации или риска параллельной модификации. Это устраняет необходимость клонировать список перед обходом или синхронизировать по списку во время обхода, оба процесса неудобны, подвержены ошибкам и сопряжены со снижением производительности. Если обходы случаются гораздо чаще, чем вставки или удаления, как часто бывает в некоторых видах ситуаций, `copyOnWritearrayList` предлагает большую производительность и более удобный доступ.

Резюме

С неизменяемыми объектами гораздо легче работать, чем с изменяемыми. Они могут быть только в одном состоянии и поэтому всегда согласованы, в силу своей природы имеют поддержку безопасности потоков, и к ним легко организовать совместный доступ. Есть множество ошибок программирования, которые легко совершить и трудно обнаружить, и которые полностью устраняются использованием неизменяемых объектов, например отсутствие синхронизации доступа между потоками или клонирования массива или объекта перед сохранением ссылки на него. При написании класса всегда стоит спросить себя, можно ли эффективно применять этот класс как неизменяемый. Возможно, вы удивитесь, как часто ответ бывает утвердительным.

Ресурсы

- Оригинал статьи: To mutate or not to mutate?
- *Обратите внимание на неизменяемость*, пункт 13 статьи Джошуа Блоха (Joshua Bloch) *Эффективное руководство по использованию языка программирования Java* (Эддисон-Уэсли, 2001), раскрывает преимущества неизменяемости более подробно.
- Изменяемые или неизменяемые? (*JavaWorld*, декабрь 2002 г.) обсуждает преимущества неизменяемости.
- Пример 63 Питера Хаггара (Peter Haggar) *Практическое руководство по использованию языка программирования Java* (Эддисон-Уэсли, 2000 г.) поощряет использование неизменяемых классов.
- Прочитайте дополнительную информацию о шаблоне Flyweight в книге "Банда четырех"; *Конструктивные шаблоны* (Эддисон-Уэсли, 1995 г.)
- Получите дополнительную информацию о классе copyOnWritearrayList и об остальной части библиотеки util.concurrent.
- В статье *Теории и практика Java* за ноябрь 2002 г. О параллелизме простыми словами (почти) предлагается введение в пакет `util.concurrent`.
- Прочитайте всю серию *Теория и практика Java*, в том числе подробности по технологиям безопасного конструирования и правильному и неправильному использованию ключевого слова `final`.
- Найдите сотни других ресурсов по Java-технологии на сайте *developerWorks* в разделе Java-технологии.

Об авторе

Брайан Гетц

Брайан Гетц (Brian Goetz) - консультант по ПО и последние 15 лет работал профессиональными разработчиком ПО. Сейчас он является главным консультантом в фирме Quiotix, занимающейся разработкой ПО и консалтингом и находящейся в Лос-Альтос, Калифорния. Следите за публикациями Брайана в популярных промышленных изданиях. Вы можете связаться с Брайаном по адресу brian@quiotix.com