3 Жажда скорости: эффективность по времени

В этой главе:

- ✓ Сравнение скорости выполнения основных структур данных, включая списки, множества и деревья.
- ✓ Оценка скорости выполнения в худшем случае и средней скорости выполнения структуры данных в долгосрочной перспективе.
- ✓ Концентрация вычислительной нагрузки в конкретном методе класса или ее распределение по всем методам.

Достижение максимально возможной скорости выполнения задачи привлекало программистов еще с доисторических времен программирования на перфокартах. В самом деле, можно сказать, что почти вся computer science подчинена удовлетворению этой потребности. В этой главе я представлю три реализации резервуара, оптимизирующие скорость разными способами. Почему три? Разве нельзя привести одну *лучшую*? Дело в том, что одной лучшей версии не существует — это один из главных выводов этой главы.

Базовые учебные курсы программирования и даже вводные курсы computer science упускают этот факт из виду. В последних широко рассматриваются вопросы эффективности по времени, особенно в алгоритмах и классах структур данных. В этих учебных курсах и учебниках рассматривается одна задача за раз, будь то обход графа или балансировка дерева. Если взять одну алгоритмическую задачу с заданным входом и желаемым выходом, можно сравнить два любых

алгоритма по скорости. Выяснится, что самым быстрым является решение с наименьшей асимптотической временной сложностью в наихудшем случае, и именно так ведется анализ для одиночных вычислительных задач.

Но реальные задачи программирования, включая наш пример с резервуарами, работают не так. Они не получают вход, вычисляют результат и завершаются. Им нужна группа взаимодействующих методов или аспектов функциональности, которые могут использоваться любое количество раз. Причем разные структуры данных могут лучше проявлять себя в одном методе и хуже — в другом, сокращая сложность первого и замедляя второй. По этой причине часто не существует однозначно лучшего решения — требуются компромиссы.

ЧАСТИЧНЫЙ ПОРЯДОК

В контексте с множеством методов (как у нас) временная сложность в наихудшем случае подразумевает *частичный порядок* между реализациями — отношение между парами элементов, при котором не каждая пара является сравнимой. Например, применим отношение «быть потомком» к парам людей. Майк и Анна принадлежат отношению, если Майк — потомок Анны. Если же два человека a и b не связаны между собой, ни пара $(a;b)$, ни пара $(b;a)$ не принадлежит отношению, то есть отношение является частичным порядком. В таких отношениях могут быть элементы, *не меньшие* любого другого (верхние элементы).

Экономисты называют такие элементы *оптимальными по Парето*. *Парето-фронтом* называется множество всех элементов, оптимальных по Парето. Если интерпретировать отношение «быть потомком» как «быть меньше», мифические Адам и Ева будут единственными верхними элементами, потому что они не меньше (то есть не являются потомками) любого другого человека.

Рассмотрим другой пример, имеющий отношение к компьютерам: правила преобразования между примитивными типами Java подразумевают отношение частичного порядка между ними. Таким образом int меньше (то есть преобразуется в) long, тогда как boolean и int несравнимы.

Если при проектировании класса нет информации, сколько раз и в какой последовательности вызывается каждый метод (нет *профиля использования*), лучшее, что можно сделать, — это выбрать реализацию, профиль скорости выполнения которой оптимален по Парето. В такой реализации никакой метод не может быть улучшен без ухудшения скорости выполнения другого метода. В этой главе представлены три реализации, оптимальные по Парето, для задачи с резервуарами.

В этой главе представлены три реализации класса резервуара, и все они соответствуют API из главы 1. Эти реализации различаются по скорости выполнения, но среди них нет такой, которая бы всегда работала быстрее других (по крайней мере в сложности наихудшего случая). Однако можно измерить *среднюю* скорость выполнения заданной реализации при выполнении длинной последовательности операций. Как показывают простые тесты из раздела 3.4, по среднему значению третья реализация быстрее других во всех сценариях, кроме самых экзотических.

НЕОЖИДАННЫЙ ВОПРОС 1

Назовите частичный порядок, существующий между классами в программе Java.

3.1. ДОБАВЛЕНИЕ ВОДЫ С ПОСТОЯННЫМ ВРЕМЕНЕМ [SPEED1]

В этом разделе я покажу, как оптимизировать метод `addWater`, имеющий линейную сложность в эталонной реализации `Reference` (глава 2). Оказывается, его сложность можно сократить до постоянного времени без повышения сложности двух других методов класса. Вряд ли можно рассчитывать на что-то лучшее.

В реализации `Reference` проблема метода `addWater` заключалась в необходимости посещения всех резервуаров, соединенных с текущим, и обновления объемов содержащейся в них воды. Это неэффективно, особенно из-за того, что *все соединенные резервуары содержат одинаковые объемы воды*. Для повышения эффективности можно вынести поле `amount` из класса `Container` в новый класс `Group`. Все резервуары, входящие в одну группу, будут ссылаться на один объект `Group`, который содержит объем воды в каждом из резервуаров группы.

Новая версия `Container`, которая называется `Speed1`, содержит одно поле:

```
private Group group = new Group(this);
```

Каждый резервуар содержит ссылку на объект нового класса `Group`, который представляет собой вложенный класс (листинг 3.1). Конструктору передается `this`, так что новая группа в начале своего существования содержит этот резервуар. Каждый объект `Group` содержит два поля экземпляра: объем воды в каждом резервуаре группы и множество всех резервуаров в группе. Таким образом, каждый резервуар знает свою группу, а каждая группа знает все входящие в нее резервуары.

Класс `Group` объявлен с ключевым словом `static`, потому что группа не должна намертво связываться с создавшим ее резервуаром. Он объявлен приватным (`private`), потому что не должен быть доступен клиентам напрямую. Так как весь класс объявлен `private`, нет смысла изменять его видимость по отношению к конструктору и полям.

Листинг 3.1. Speed1: вложенный класс Group

```
private static class Group {
   double amountPerContainer;
   Set<Container> members;   ❶ Множество всех соединенных резервуаров

   Group(Container c) {
      members = new HashSet<>();
      members.add(c);
   }
}
```

На рис. 3.1 изображена ситуация после выполнения первых трех частей основного сценария использования. Напомню, что эти три части создают четыре резервуара (a, b, c и d) и вызывают следующие методы:

```
a.addWater(12);
d.addWater(8);
a.connectTo(b);
b.connectTo(c);
```

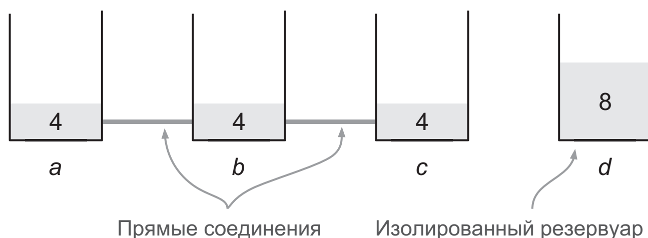

Рис. 3.1. Ситуация после выполнения первых трех частей основного сценария. Резервуары от *a* до *c* соединены, а в *a* и *d* наливается вода

Метод `connectTo` очень похож на одноименный метод из эталонной реализации `Reference` (https://bitbucket.org/mfaella/exercisesinstyle).

На рис. 3.2 изображено распределение памяти `Speed1` на данный момент сценария использования. Так как резервуары соединены в две группы, существуют два объекта типа `Group`, каждый из которых содержит ссылку на множество всех резервуаров, принадлежащих группе, а также объем воды в каждом из этих резервуаров.

Методы чтения и записи `Container` работают непосредственно с объектом `Group`:

Листинг 3.2. Speed1: методы getAmount и addWater

```
public double getAmount() { return group.amountPerContainer; }

public void addWater(double amount) {
   double amountPerContainer = amount / group.members.size();
   group.amountPerContainer += amountPerContainer;
```

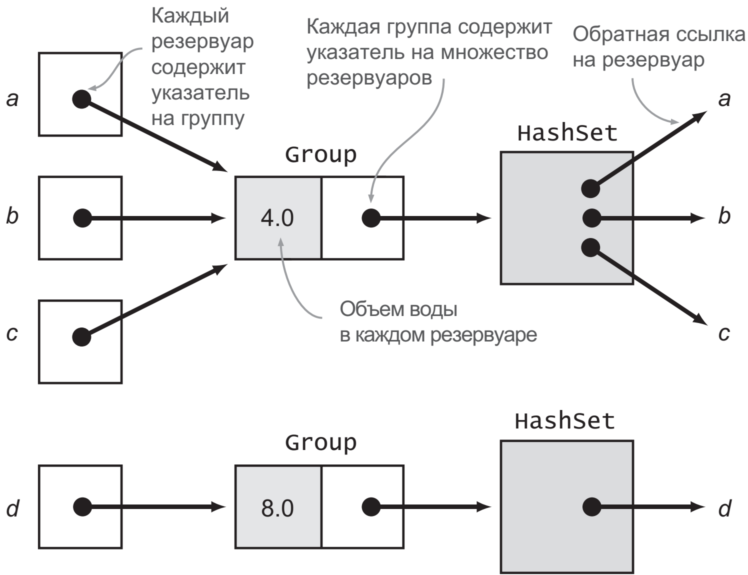

Рис. 3.2. Распределение памяти реализацией Speed1 после выполнения первых трех частей основного сценария

3.1.1. Временная сложность

По аналогии с `Reference`, метод `connectTo` все равно должен перебрать все резервуары в группе, что приведет к временным сложностям (табл. 3.1). Как видите, «узким местом» реализации является метод `connectTo`.

Таблица 3.1. Временные сложности для Speed1, где n обозначает общее количество резервуаров

Метод	Временная сложность
`getAmount`	$O(1)$
`connectTo`	$O(n)$
`addWater`	$O(1)$

Два действия в методе `connectTo` требуют линейного времени для завершения:

1. Слияние элементов двух групп методом `addAll`.
2. Уведомление элементов одной из объединяемых групп о том, что их группа изменилась.

Первый шаг легко заменяется более быстрым альтернативным решением. Перейдите от множеств на связанные списки, и *вуаля*: слияние двух коллекций

превратится в операцию с постоянным временем. Избежать второго шага намного сложнее. Собственно, невозможно заставить `connectTo` выполняться за постоянное время без повышения временной сложности `getAmount`. Но если вам непременно понадобится `connectTo` с постоянным временем, примените реализацию из следующего раздела.

3.2. ДОБАВЛЕНИЕ СОЕДИНЕНИЙ ЗА ПОСТОЯННОЕ ВРЕМЯ [SPEED2]

Цель этого раздела — уменьшение сложности `connectTo` до постоянного времени, которое приведет к новой версии класса резервуара — `Speed2`. Для этого мы используем два приема:

1. Представим группы соединенных резервуаров в виде структуры данных, которая поддерживает операцию слияния с постоянным временем.
2. Максимально задержим обновление объема воды.

Для начала нам понадобится принципиально новый способ представления группы соединенных резервуаров: реализованный вручную циклический связанный список.

3.2.1. Представление групп в виде циклических списков

Циклический связанный список представляет собой последовательность узлов, в которой каждый узел содержит ссылку на следующий узел с циклическим замыканием списка. В списке нет ни первого узла, ни последнего. Пустой циклический связанный список не содержит ни одного узла, тогда как в списке из одного узла этот узел указывает сам на себя как на следующий узел.

В приложении с резервуарами каждый резервуар представляет собой узел односвязного циклического списка, который содержит поле `amount` и одну ссылку `next`:

Листинг 3.3. Speed2: поля

```
public class Container {
   private double amount;
   private Container next = this;
```

НЕОЖИДАННЫЙ ВОПРОС 2

С какой сложностью выполняется удаление узла из односвязного циклического списка?

У циклических связанных списков есть одно важное свойство, из-за которого они так популярны: если у вас есть два произвольных узла из двух таких списков, то слияние этих списков выполняется за постоянное время (даже если списки являются односвязными) посредством перестановки указателей `next` двух узлов (рис. 3.3).

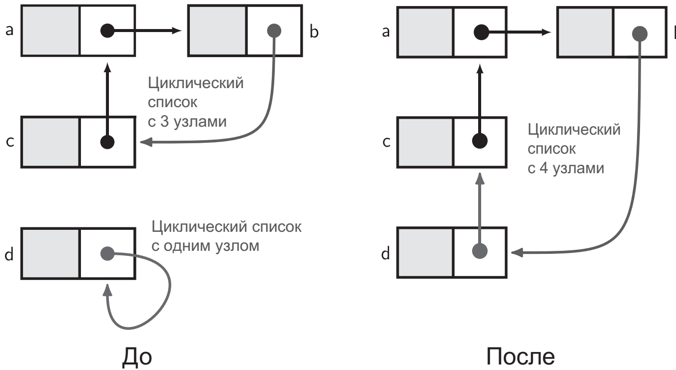

Рис. 3.3. Перестановка указателей next двух узлов (b и d) приводит к слиянию двух циклических связных списков

Тем не менее этот метод работает только в том случае, если два узла принадлежат разным спискам. Иначе перестановка ссылок ведет к обратному эффекту: список разбивается на два разных списка. А значит, эта реализация страдает от того же ограничения, которое упоминалось для `Novice`: `connectTo` работает правильно, только если соединяемые резервуары еще не соединены — даже косвенно.

Напрашивается вывод, что метод `connectTo` должен *проверять*, соединены ли два резервуара, прежде чем пытаться соединять их. Но для этого необходимо просканировать по крайней мере один из двух списков, а эта операция уже не выполняется с постоянным временем. Для достижения желаемой скорости выполнения с постоянным временем приходится мириться с подобным снижением защищенности программы. Положение будет исправлено в главе 5, когда мы займемся построением классов резервуаров с повышенной надежностью.

СОВЕТ

Стандартная реализация связанных списков на Java (LinkedList) не поддерживает конкатенацию с постоянным временем. Вызов `list1.addAll(list2)` перебирает все элементы `list2`.

КАК НАСЧЕТ ОБЫЧНОГО СВЯЗАННОГО СПИСКА?

Циклические списки — не единственная структура данных, обеспечивающая слияние за постоянное время. Обычный связанный список также обладает таким свойством, при условии что операция слияния может напрямую обращаться к первым и последним элементам двух списков («голове» и «хвосту»). Представьте, что вы можете напрямую обратиться к полям `head` и `tail` двух непустых связанных списков `list1` и `list2`. Слияние их посредством конкатенации выполнится следующими строками:

```
list1.tail.next = list2.head;
list1.tail = list2.tail;
```

В результате `list1` представит результат слияния, а `list2` не изменится.

Тем не менее связанные списки не удастся использовать для соединения резервуаров с водой за постоянное время. Чтобы понять почему, вспомните, что каждый резервуар должен иметь прямой доступ к первому и последнему элементу списка. При слиянии двух групп необходимо обновить все задействованные резервуары, чтобы они отражали новые значения `head` и `tail` после слияния. Такое обновление требует линейного времени.

На рис. 3.4 представлено распределение памяти в два момента во время выполнения основного сценария использования, с реализацией резервуаров из этого раздела (то есть `Speed2`). Как видите, структура точно совпадает с рис. 3.3, не считая того, что узлы в списках теперь представляют резервуары с водой.

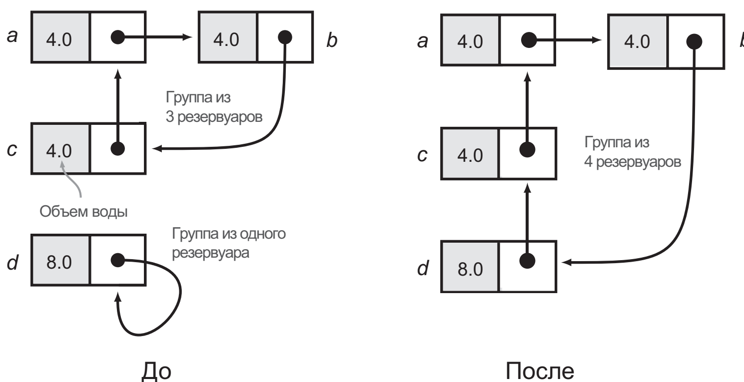

Рис. 3.4. Распределение памяти Speed2 при выполнении основного сценария использования, до и после b.connectTo(d). Перестановка указателей next объектов *b* и *d* приводит к слиянию двух групп в одну