5. Примеры программных ошибок

Здесь приводятся примеры программных ошибок из практики, приводящих к очень изощренным сбоям.

Всё подряд

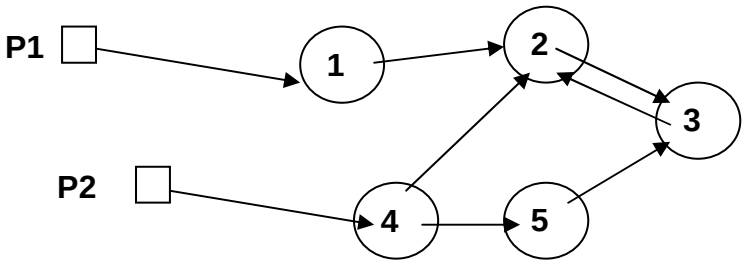
Сериализация в Java. Лучше 40 раз по разу, чем 1 раз 40 раз

Ошибка, причина которой – незнание некоторых нюансов стандартного механизма сериализации в Java. В лабораторной работе два экземпляра приложения передавали друг другу через сокет сериализуемые объекты, которые отображались в панельке как движущиеся шарики, звездочки и т.п. При кликаньи мышью по объекту в одном приложении его копия появлялась в том же месте в панельке другого. Движение каждого объекта управлялось отдельным потоком – при десериализации ссылка на новый объект записывалась в вектор и передавалась вновь создаваемому потоку. Эффект обнаружился при повторном клике по одному и тому же объекту – **вместо создания новой копии увеличивалась скорость движения старой, а новая вообще не появлялась**. Сначала грешили на синхронизацию, а потом возникла стойкая догадка: увеличение скорости может быть связано только с тем, что несколько потоков разделяют один и тот же объект. На обнаружение причины и устранение дефекта понадобилось 2 часа.

Изучаем матчасть. При сериализации связанных объектов имеет место граф сериализации. Сериализуется объект, если он содержит ссылки или массивы ссылок на сериализуемые объекты, то они сериализуются рекурсивно. При этом сериализуемые объекты помечаются, что исключает зацикливание при обходе графа.

Кроме того, для сериализуемых объектов создаются уникальные идентификаторы, которые необходимы для восстановления ссылок при десериализации. Т.е. при первой сериализации объект передается «по полной программе» и для него создается внутренний идентификатор, если алгоритм повторно «натыкается» на помеченный объект, то передается только его идентификатор.

Такая схема вполне логична, если в поток сериализуется связанная структура данных по частям, например, по ссылке p1 сериализуются объекты 1,2,3, а по ссылке p2 – 4,5, при переходе по ссылкам 4-2 и 5-3 в поток пишутся только идентификаторы объектов. Таким образом, пометка объектов в графе сериализации **связана не с отдельным вызовом метода writeObject, а с потоком в целом.**



Попытка сериализовать **последовательность состояний объекта** приведет к тому, что поток «заклинит» на первом состоянии, которое на приемном конце (если речь идет о сети) не будет обновляться.

Для воспроизведения ситуации написал простенький тест. Объект 10 раз серилизуется в файл, каждый раз увеличивая внутреннее значение от 0 до 10. При десериализации получаем в массиве 10 ссылок на единственный объект.

```
package javaserializationeffect;
import java.io.*;

/*   Один и тот же объект - содержимое меняется, сериализуется ID */

public class JavaSerializationEffect {

   public void effect() throws Throwable{
      ObjectOutputStream out=new ObjectOutputStream(new FileOutputStream("test.dat"));
      Test xx=new Test();
      for(int i=0;i<10;i++){
         xx.setA(i);
         out.writeObject(xx);
         out.reset();                  // Сброс графа сериализации в потоке !!!!!!!!!!!!!!
         }                             // Без этого объект пишется 1 раз, а после него - ссылки
      out.close();
      Test zz[]=new Test[10];
      ObjectInputStream in=new ObjectInputStream(new FileInputStream("test.dat"));
      for(int i=0;i<10;i++)
         zz[i]=(Test)in.readObject();
      in.close();
      for(int i=0;i<10;i++)
         System.out.println("a="+zz[i].getA() + " "+zz[i]);
      }
   public static void main(String[] args) {
      try {
         JavaSerializationEffect xx=new JavaSerializationEffect();
         xx.effect();
         } catch(Throwable ee){ System.out.println(ee); }
   }
}
```

Решение проблемы. Метод ObjectInputStream.**reset** сбрасывает граф сериализации потока. В тесте это приводит к корректной записи последовательности состояний объекта и ее восстановлении в виде массива ссылок на объекты с уникальными значениями. Последующие тест с двумя потоками показал, что графы сериализации потоков независимы, т.е. отметка объектов в одном потоке никак не сказывается на сериализации в другом (вероятно, поток поддерживает вектор ссылок сериализованных объектов).

Замечание. Если сериализовать структуру данных, изображенную на рисунке последовательностью вызовов writeObject(p1), reset(),writeObject(p2), то при десериализации получим 2 несвязанные структуры данных p1-1,2,3, p2-4,5,2*,3*, т.е. объекты 2 и 3 будут десериализованы в 2 экземплярах, а остальные – в одном.

Замечание. Чтобы тест соответствовал заголовку, замените константу 10 на 40 (шутка).

Резюме. При использовании сторонних компонент желательно знать не только их интерфейс, но и внутренние механизмы их функционирования.

Странное влияние перезагрузки планшета

Характер сбоя. При запуске программы в планшетном ПК сразу после перезагрузки она, как правило, зависала. Повторный запуск происходил нормально.

Причина ошибки. При запуске планшета загрузка профиля пользователя и его настройка выполнялись в фоновом потоке, в основном потоке открывалось окно с заставкой и выполнялся периодический опрос по таймеру логической переменной. Когда настройка профиля завершалась, поток устанавливал эту логическую переменную.

Установка логической переменной завершения производилась в одном из вызываемых в фоновом потоке методов (не было удалено), в результате чего основной

поток получал уведомление о завершении настройки раньше, чем это происходило на самом деле. Но при низкой загрузке процессора фоновый поток успевал выполнить завершающие действия до следующего интервала таймера. При инициализации ОС во время перезагрузки скорость потока была ниже, в результате чего он не успевал.

Решение проблемы. Удаление лишней команды.

Пробуксовка сервера

Постановка задачи. Сервер сбора статистики осуществляет опрос сервера общественного транспорта и получает оттуда данные о состоянии транспортных средств (бортов) – координаты, скорость. Интервал изменения данных о состоянии борта – 1.5 – 2 мин. Количество бортов может достигать порядка 1000.

Сервер собирает статистику о скорости движения на каждом сегменте дорожной сети – всего около 8000 (сегменты каждого маршрута и транспортной сети в целом). Статистика собирается отдельно для каждого часа. Итого в течение недели набирается 18*7*8000 около 1 млн. объектов статистики - записей БД.

Механизмы сервера. Сервер использует механизмы загрузки записей по требованию и отложенной записи. Каждый сегмент имеет набор ссылок на ячейки статистики. Первоначально все ссылки нулевые. При добавлении данных в некоторую ячейку при отсутствии ее в памяти инициализируется загрузка из БД, а текущие данные сохраняются во временном объекте. По событию «окончание загрузки» данные из загруженной ячейки и временного объекта сливаются.

Загрузка и обновление записей осуществляются при помощи паттерна «планировщик» (scheduler), который организует очередь запросов и исполняет их код последовательно в одном потоке.

Имеется фоновый поток обновления данных в БД. Каждая измененная ячейка помечается специальным признаком и получает последовательный номер изменения. Фоновый поток выбирает заданный процент наиболее «старых» измененных ячеек и записывает их содержимое в БД.

Фоновый поток также удаляет заданный процент наиболее «старых» по времени обращения ячеек, тем самым происходит автоматическая разгрузка памяти сервера от неиспользуемых данных.

Все это представляет собой аналогию механизма виртуальной памяти – загрузка страниц виртуального адресного пространства по требованию, вытеснение страниц.

Характер сбоя. При высокой интенсивности движения возникал процесс лавинообразного увеличения длины очереди на чтение/обновление, что было видно по количеству элементов списка - запросов и объектов – адаптеров обратного вызова. В конце концов это приводило к переполнению динамической памяти (кучи).

Высказывались разные фантастические предположения вплоть до утечек памяти для объектов анонимных классов в Scala (первоначально сервер был написан на этом языке), или недостаточной производительности сервера БД MySQL.

Ошибка и особенности ее появления. Дефект состоял в том, что ячейки, для которых, инициализировался процесс загрузки или фоновый поток выполнял обновление, никак не отмечалась. Если сервер стартовал во время низкого трафика, то сбой не возникал. При высоком трафике во время старта сервера формировалось одновременно много запросов на загрузку ячеек (порядка 1000). Следует заметить, что финальная часть запроса выполняется планировщиком в основном потоке и, поэтому, также происходит не сразу (образуется неявная очередь завершений). В результате те ячейки, которые не успевали загрузиться или обновиться за текущий цикл сервера, повторно попадали в очередь

запросов несмотря на то, что соответствующие процессы были запущены. Возникал эффект «снежного кома» или пробуксовки: чем больше была длина очереди, тем большее количество ячеек на успевало обслужиться и попадало повторно в эту же очередь.

Решение проблемы. В каждый объект была введена логическая переменная, обозначающая, что он уже находится в процессе обновления. При старте сервера загружаются сразу все объекты статистики текущих суток. В работе сервера имеется перерыв с 0 до 6 часов, в начале которого текущая статистика выгружается.

Почему перестал работать двоичный поиск

Пример дефекта формата входных данных. Система управления земельным кадастром использовала БД координат объектов, в котором каждому объекты был присвоен уникальный ID, генерируемый в порядке записи объектов в таблицу. Затем данные экспортировались в два двоичных файла – координатных записей переменной длины и файл, содержащий пары ID-адрес записи в первом файле. При поиске объекта по ID программа-«рисовалка» искала объект в таблице, загруженной в память, обычным двоичным поиском.

От одного из клиентов поступило сообщение, что «рисовалка» стало как-то странно случайным образом пропускать объекты при их изображении, хотя в базе они достоверно присутствовали. Не буду описывать прелести дистанционной отладки по межгороду по базе клиента (дело происходило в 90-х годах). Причина сбоя оказалась следующей. При восстановлении «сломанной» базы из архивов путем ручного копирования записей, куски таблиц были перепутаны, и исходный порядок возрастания нарушился, хотя все записи были восстановлены. А поскольку компонента экспорта БД в двоичный файл открывала физическую таблицу, это привело к нарушению порядка возрастания ID-ов в двоичном файле. В результате двоичный поиск при делении пополам вел себя непредсказуемо. Для исправления дефекта понадобилось заменить обращение к физической таблице SQL-запросом с ORDER BY ID.

Ошибки и дефекты реактивности и производительности

Собственно, любое прямолинейное решение задачи может оказаться дефектом, если оно не устраивает по производительности или реактивности для тех объемов данных, с которыми сталкивается программа. Исправление дефекта – поиск более «умного» или оригинального решения – не является простой технической задачей.

Пресловутые String и StringBuffer в Java

Простой пример оценки производительности Java при работе с классами String и StringBuffer.

```
   public void experience1(){
      long t1,t0=new Date().getTime();
      String ss="";
      for(int i=0;i<10000000;i++){
         ss+="a";
         if (i%10000==0){
            t1=new Date().getTime()-t0;
            System.out.println("n="+i+" time="+t1+" мс
mem="+Runtime.getRuntime().totalMemory()/1000+ "Кб");
            if (t1>dt) break;
            }
         }
      }
   public void experience2(){
      long t1,t0=new Date().getTime();
```

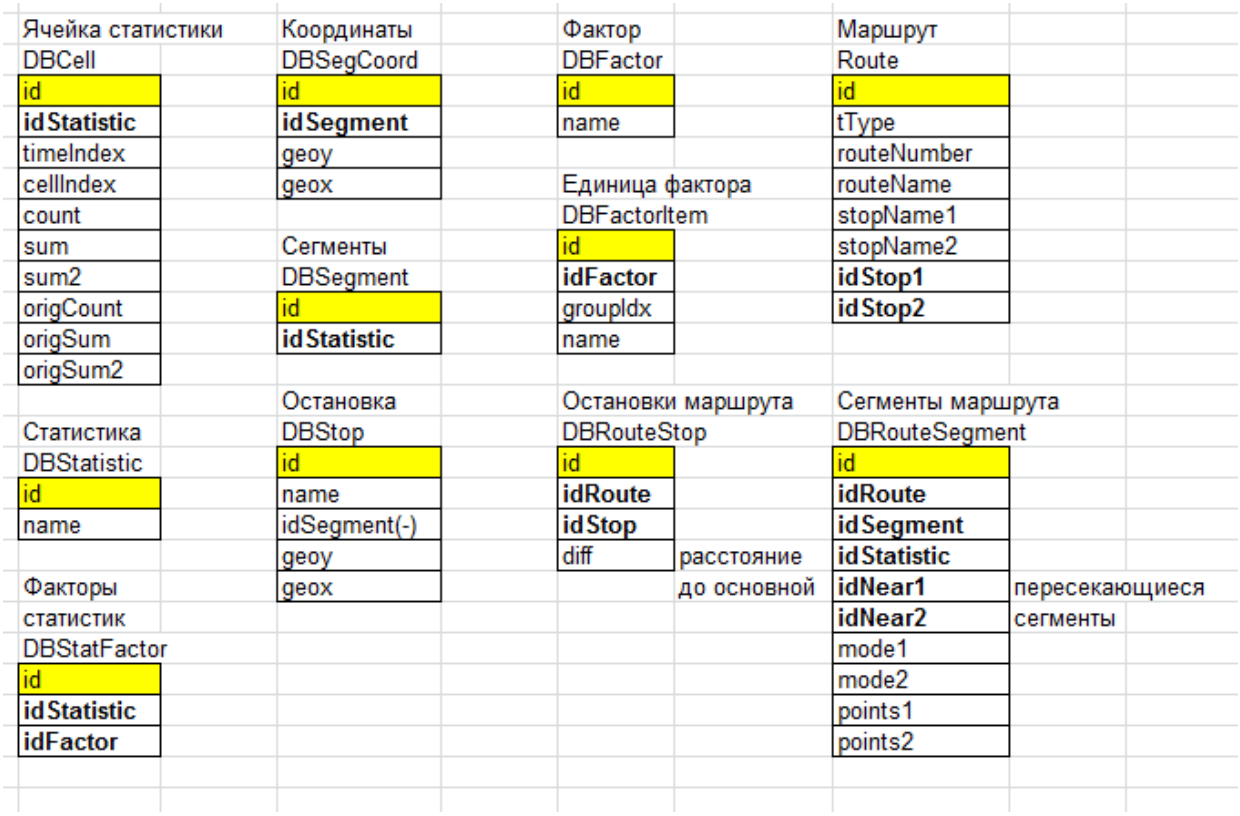
```
        StringBuffer sb=new StringBuffer();
        for(int i=0;i<20000000;i++){
            sb.append("a");
            if (i%1000000==0){
                t1=new Date().getTime()-t0;
                System.out.println("n="+i+" time="+t1+" мс
mem="+Runtime.getRuntime().totalMemory()/1000+ "Кб");
                if (t1>dt) break;
                }
            }
        }
```

Сравнение результатов:

- для StringBuffer - **n=37000000 time=1130 мс mem=187580Кб –** использование памяти – около **5** байт на символ, в мусоре старые динамические массивы, наращиваемые в геометрической прогрессии, производительность – **0.03** мкс/символ;
- для String - **n=230000 time=63490 мс mem=16318Кб –** использование памяти – около **71** байт на символ, значительное количество мусора за счет старых объектов. производительность – **276** мкс/символ;

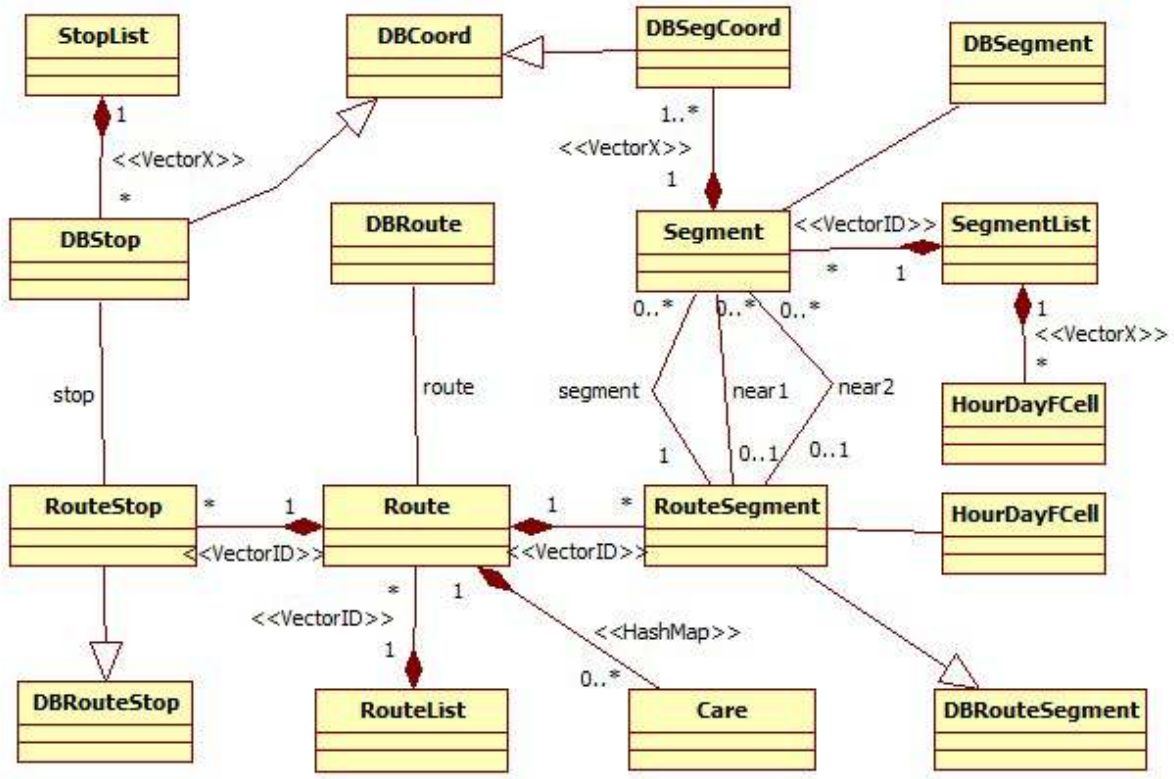
Не откладывай на БД, то, что можешь сделать в памяти

Постановка задачи. Транспортная сеть представляет собой множество маршрутов, остановок и сегментов (отрезков пути). Каждый маршрут содержит вектора ссылок на остановки и сегменты, которые ему принадлежат. В БД эта структура выглядит следующим образом.



Таблицы связей **DBRouteStop** и **DBRouteSegment** содержат пары ключей, соответствующие связям между сущностями маршрут-остановка и маршрут-сегмент. Последовательность сегментов и остановок в маршруте «естественно историческая»: записи пишутся при импорте в порядке их следования в маршруте. Диаграмма классов

транспортной сети содержит DAO-классы БД и классы бизнес-объектов, представляющие сущности транспортной сети в памяти (на уровне бизнес-логики).



Бизнес-объекты **RouteList, SegmentList и StopList** содержат вектора ссылок на объекты – маршруты, сегменты и остановки. Бизнес-объект **Route –** маршрут содержит вектора объектов **RouteStop и RouteSegment** со ссылками на соответствующие объекты **DBStop и DBSegment.**

Дефект производительности. Загрузка транспортной сети, содержащей 160 маршрутов, 670 остановок и 1850 сегментов составляет 40 сек. Львиная доля времени уходит на исполнение 2*160 = 320 запросов для получения связанных объектов **DBRouteStop и DBRouteSegment** для каждого маршрута.

Решение проблемы. «Ручное» связывание с использованием хэш-таблиц. Таблицы **DBRoute, DBStop, и DBSegment** грузятся в виде векторов объектов, для каждого из которых создается хэш-таблица с ключем – идентификтором в БД (включено в функционал класса **VectorID**). Затем грузятся таблицы **DBRouteStop и DBRouteSegment** целиком в вектора объектов. Далее в циклах по каждой таблице по паре ключей ищутся пары объектов **Route(DBRoute) – DBStop, Route(DBRoute) – DBSegment** и в вектор найденного объекта **Route** добавляется создаваемые **RouteStop и RouteSegnment** со ссылками на оригинальные бизнес-объекты **Stop и Segment.** Таким образом, связывание происходит одновременно по всем маршрутам не средствами SQL-запросов, а путем поиска нужных объектов по ключу в хэш-таблицах.

Резюме. Время загрузки структуры данных из БД уменьшается от 40 сек. до 2-3 сек.

Не вычерпывай бочку чайной ложкой

Дефект производительности, связанный с генерацией потока мелких запросов. Сервер сбора статистики опрашивает сервер источника данных (городской транспорт) и накапливает статистику по скорости движения транспортных средств на сегментах сети (в целом по сегменту и для каждого маршрута, всего около 8000 элементов - ячеек

статистики). Для каждого элемента каждый час в БД (MySQL) вносится новая запись (при наличии накопленных данных). Необходимо оперативно сбрасывать в БД все изменения ячеек статистики при получении данных от любого транспортного средства (до 2000 бортов с интервалом обновления данных – 2 мин).

Первоначальная идея состояла в формировании потока SQL-запросов INSERT/UPDATE на обновление записей в БД, причем был реализован механизм отложенной записи. Отдельный поток каждые 10 сек. Просматривал все ячейки и формировал запросы для всех, отмеченных как измененные. Хотя процессы опроса (сбора) данных и обновления БД были разнесены (не мешали друг другу), SQL-сервер эпизодически не справлялся с таким потоком запросов.

Решение проблемы. Использование SQL-запросов REPLACE, которые допускают множественные обновления записей. Запрос может как обновлять записи, так и добавлять их при отсутствии совпадения. Однако простой группировкой данных дело не заканчивается. При исполнении SQL-запроса список в качестве результата получается массив первичных ключей для добавляемых записей, который надо «распихивать» по исходным DAO-объектам в памяти.