MAC-211 — Laboratório de Programação

Exercício Programa 3: JBase - Gerenciador de Arquivos

Entrega: 25 de maio de 1999

Nesta segunda etapa do projeto do JBase o objetivo é chegar a um pacote que possa ser utilizado por aplicações. Para isso, temos que definir a API[1] do JBase. As duas primeiras seções deste texto fazem isso. A primeira dá uma visão rápida da utilização do pacote. A segunda especifica as interfaces e classes da API do JBase. Em vez de fazer uma "leitura sequencial" dessas duas seções, você provavelmente achará melhor ir consultando a seção 2 à medida que for lendo os exemplos da seção 1. As seções restantes especificam aspectos da implementação do JBase, do teste do pacote e da entrega deste EP.

1 Utilização do JBase

1.1 Objetos Armazenáveis

Programas de aplicação usam o JBase para guardar objetos em bancos de dados. Os objetos armazenáveis pelo JBase precisam satisfazer três condições:

1. saber "se escrever" num buffer do JBase.
2. saber "se ler" de um buffer do JBase;
3. saber informar quantos bytes ele ocupa quando escrito num buffer.

Essas condições correspondem aos três métodos da interface `DBObject`. (Veja na página 6 a definição dessa interface.) Para que instâncias de uma classe sejam armazenáveis num banco de dados, a classe deve implementar a interface `DBObject`. Eis um exemplo de classe cujas instâncias são armazenáveis:

```
import br.usp.ime.jbase.DBObject;
import br.usp.ime.jbase.Buffer;

class Aluno implements DBObject {

    // campos (só dois, por simplicidade)

    int numUSP;     // campo de comprimento fixo (4 bytes)
    String nome;    // campo de comprimento variável
```

[1]A API (*application programming interface*, ou interface de programação de aplicações) de um sistema define como os programas de aplicação interagem com o sistema. No caso de um pacote Java, a API é o conjunto de interfaces e classes públicas do pacote.

```java
    // construtor

    Aluno(int numUSP, String nome) {
        this.numUSP = numUSP;
        this.nome = nome;
    }

    // construtor sem argumentos (importante)

    Aluno() {}

    // métodos específicos de um Aluno

    ...

    // métodos da interface DBObject

    public int writeTo(Buffer buf, int pos, int maxLen)
            throws UTFDataFormatException {
        int utfCount;       // número de bytes do nome escritos no buffer

        if (maxLen < 4) {
            return 0;                   // o espaço não dá nem para o numUSP
        }
        buf.writeInt(pos, numUSP);      // escreve 4 bytes começando em pos
        utfCount = buf.writeUTF(pos + 4, maxLen - 4, nome);
        return (utfCount == 0) ? 0 : 4 + utfCount;
    }

    public void readFrom(Buffer buf, int pos)
            throws UTFDataFormatException {
        numUSP = buf.readInt(pos);      // lê 4 bytes a começando em pos
        nome = buf.readUTF(pos + 4);    // agora lê a partir de pos + 4
    }

    public int getSize() {
        return 4 + Buffer.getSizeUTF(nome);
    }

}
```

O método `writeTo` tenta escrever no buffer `buf` todos campos do `Aluno`, começando na posição `pos` e usando no máximo `maxLen` bytes do buffer. Ele retorna o número de bytes escritos, ou zero caso os campos não tenham sido completamente escritos por precisarem de mais de `maxLen` bytes. O método `readFrom` lê do buffer `buf` todos campos do `Aluno`, começando a leitura na posição `pos` do buffer. O método `getSize` retorna o espaço (em bytes) que os campos do `Aluno` ocupam quando escritos num buffer.

O programador de uma aplicação precisa escrever métodos `writeTo`, `readFrom` e `getSize` para toda classe cujas instâncias ele vai armazenar num banco de dados do JBase. Esse é o tipo de tarefa que deveria ser automatizada: é chata e sujeita a erros, porém absolutamente trivial. Se fôssemos construir uma "versão profissional" do JBase, nela deveríamos incluir uma ferramenta que automatizasse a geração dos métodos `writeTo`, `readFrom` e `getSize`.

1.2 Armazenando Objetos

Eis um fragmento de programa que mostra como uma aplicação chama métodos da API do JBase para criar um banco de dados, criar uma coleção (inicialmente vazia) de registros dentro do banco de dados, e adicionar registros a essa coleção:

```
// cria um banco de dados com o nome MeuDB
Database db = Database.create("MeuDB", 2000);

// cria coleção de registros com o nome Alunos
RecordSet alunos = db.createRecordSet("Alunos", Aluno.class);

// adiciona um registro à coleção
alunos.add(new Aluno(3030303, "José da Silva"));

// adiciona outro registro
Aluno a = new Aluno(5050505, "Maria de Souza");
alunos.add(a);
```

Cada registro do `RecordSet` corresponde a um objeto armazenado. Um `RecordSet` é uma coleção homogênea, isto é, todos os objetos nela armazenados devem ser instâncias de uma mesma classe, especificada na criação do `RecordSet` (pelo segundo parâmetro do método `createRecordSet`).

O seguinte fragmento de programa ilustra a adicão de registros à uma coleção já existente:

```
// acessa o banco de dados com o nome MeuDB
Database db = Database.instance("MeuDB");

// pega a coleção de registros com o nome Alunos
RecordSet alunos = db.getRecordSet("Alunos");

// adiciona registros à coleção
alunos.add(new Aluno(6060606, "Ana Luiza Lemos"));
alunos.add(new Aluno(7070707, "João Pereira"));
```

1.3 Recuperando Objetos

Os métodos `retrieveAll`, `exactMatchQuery` e `rangeQuery` recuperam objetos armazenados num `RecordSet`. O primeiro é usado para obter todos os objetos, o segundo para buscar os objetos que tem um de seus campos satisfazendo uma condição de igualdade (por exemplo: obter todos os alunos com `nome` igual a `"José da Silva"`), e o terceiro para buscar os objetos que tem um de seus campos satisfazendo uma condição de pertinência a intervalo (por exemplo: obter todos os alunos com `numUSP` entre `3000000` e `4000000`). Para poderem retornar uma subcoleção potencialmente muito grande dos objetos armazenados num `RecordSet`, esses três métodos retornam um `DBIterator`. (Veja na página 6 a definição da interface `DBIterator`.) O seguinte trecho de programa, que imprime os nomes de todos os alunos armazenados no `RecordSet` referenciado pela variável `alunos`, ilustra o uso de um `DBIterator`:

```
DBIterator it = alunos.retrieveAll();
Aluno a = new Aluno(); // os campos deste objeto serão modificados pelas
                       // chamadas it.next(a) efetuadas na linha abaixo
while (it.next(a) != DBIterator.END) {
    System.out.println(a.nome);
}
```

O iterador `it` representa uma seqüência de objetos, mas não materializa (coloca na memória) toda a seqüência de uma vez. Ele tem um método `next`, que fornece o próximo elemento da seqüência. Cada chamada a `it.next(a)` copia para os campos de `a` os campos correspondentes do próximo elemento da seqüência e retorna o record id desse elemento. Caso o próximo elemento não exista, `it.next(a)` retorna `DBIterator.END`.

Para imprimir a lista de números USP dos alunos que tem `nome` igual a `"José da Silva"` (podem haver muitos xarás), uma aplicação chamaria `exactMatchQuery` em vez de `retrieveAll`:

```
DBIterator it = alunos.exactMatchQuery("nome", "José da Silva");
Aluno a = new Aluno();
while (it.next(a) != DBIterator.END) {
    System.out.println(a.numUSP);
}
```

Para obter os alunos com `numUSP` entre `3000000` e `4000000`, a aplicação usaria um `rangeQuery`:

```
DBIterator it =
    alunos.rangeQuery("numUSP", new Integer(3000000), new Integer(4000000));
Aluno a = new Aluno();
while (it.next(a) != DBIterator.END) {
    System.out.println(a.nome);
}
```

A interface `DBIterator` tem também métodos `hasNext` e `nextRecordId`. O primeiro retorna um `boolean` que indica se o próximo objeto existe ou não. O segundo retorna o record id do próximo objeto, sem retornar o objeto. Tendo o record id, pode-se obter o objeto chamando o método estático `JBase.retrieve`. Com essas informações, podemos reescrever o último trecho de programa da seguinte maneira:

```
DBIterator it =
    alunos.rangeQuery("numUSP", new Integer(3000000), new Integer(4000000));
Aluno a = new Aluno();
while (it.hasNext()) {
    long rid = it.nextRecordId()
    JBase.retrieve(rid, a);
    System.out.println(a.nome);
}
```

O mesmo trecho poderia também ser escrito assim:

```
DBIterator it =
    alunos.rangeQuery("numUSP", new Integer(3000000), new Integer(4000000));
Aluno a = new Aluno();
```

```
long rid;
while ((rid = it.nextRecordId()) != DBIterator.END) {
    JBase.retrieve(rid, a);
    System.out.println(a.nome);
}
```

Embora esta seção seja sobre a utilização (e não sobre a implementação) do JBase, abrimos um parêntese para observar que é trivial escrever o método `next` em termos dos métodos `nextRecordId` e `JBase.retrieve`:

```
// numa classe que implementa a interface DBIterator:
public long next(DBObject obj)
        throws WrongTypeException,
               java.io.IOException, java.io.UTFDataFormatException {
    long recId = nextRecordId();
    if (recId != DBIterator.END) {
        try {
            JBase.retrieve(recId, obj);
        }
        catch (InvalidRecordIdException e) {
            throw new Error("Erro interno. Exceção inesperada: \n"
                            + e.getMessage());
        }
    }
    return recId;
}
```

1.4 Removendo Objetos

O método estático `JBase.remove` recebe um record id e remove o registro correspondente. O seguinte trecho de código remove os registros dos alunos que tem `numUSP` igual a `2020202` (é de se esperar que não exista mais de um aluno satisfazendo essa condição):

```
DBIterator it = alunos.exactMatchQuery("numUSP", new Integer(2020202));
while (it.hasNext()) {
    JBase.remove(it.nextRecordId());
}
```

1.5 Atualizando Objetos

Para atualizar um ou mais campos de um objeto armazenado num `RecordSet`, chame o método `update` sobre o `RecordSet`. O exemplo seguinte altera para `4040404` o `numUSP` dos alunos (presumivelmente há no máximo um) armazenados com `numUSP` igual a `404`:

```
DBIterator it = alunos.exactMatchQuery("numUSP", new Integer(404));
Aluno a = new Aluno();
long rid;
while ((rid = it.next(a)) != DBIterator.END) {
    a.numUSP = 4040404;
    alunos.update(rid, a);
}
```

2 A API do JBase

A API do JBase é composta pelas interfaces e classes especificadas nesta seção, mais a classe `Buffer` e as várias classes de exceção.

2.1 Interface `DBObject`

```
package br.usp.ime.jbase;

public interface DBObject {

    public abstract int writeTo(Buffer buf, int bufPos, int maxLen)
            throws java.io.UTFDataFormatException;

    public abstract void readFrom(Buffer buf, int bufPos)
            throws java.io.UTFDataFormatException;

    public abstract int getSize();

}
```

2.2 Interface `DBIterator`

```
package br.usp.ime.jbase;

public interface DBIterator {

    long END = -1L;

    public abstract boolean hasNext()
            throws java.io.IOException, java.io.UTFDataFormatException;

    public abstract long next(DBObject obj)
            throws WrongTypeException,
                   java.io.IOException, java.io.UTFDataFormatException;

    public abstract long nextRecordId()
            throws java.io.IOException, java.io.UTFDataFormatException;

}
```

2.3 Interface `RecordSet`

```
package br.usp.ime.jbase;

public interface RecordSet {

    public abstract void add(DBObject obj)
            throws NotEnoughSpaceException, WrongTypeException,
                   java.io.IOException, java.io.UTFDataFormatException;
```

```
    public abstract DBIterator retrieveAll()
            throws java.io.IOException;

    public abstract DBIterator exactMatchQuery(String keyName,
                                               Object keyValue)
            throws java.io.IOException, java.io.UTFDataFormatException;

    public abstract DBIterator rangeQuery(String keyName,
                                          Object lowKeyValue,
                                          Object highKeyValue)
            throws java.io.IOException, java.io.UTFDataFormatException;

    public abstract void update(long recordId, DBObject obj)
            throws InvalidRecordIdException, WrongTypeException,
                   NotEnoughSpaceException, java.io.IOException;

    public abstract void setFillFactor(float f)
            throws InvalidFillFactorException;

    public abstract float getFillFactor();

}
```

2.4 Classe Database

```
package br.usp.ime.jbase;

public class Database {

    public static Database create(String name, long nBlks)
            throws AlreadyExistsException, java.io.IOException {
        ...
    }

    public static Database instance(String name)
            throws java.io.FileNotFoundException, java.io.IOException {
        ...
    }

    public void close() throws java.io.IOException {
        ...
    }

    public RecordSet createRecordSet(String name, Class recordType)
            throws AlreadyExistsException, NotEnoughSpaceException,
                   WrongTypeException, java.io.IOException {
        ...
    }
```

```
    public RecordSet getRecordSet(String name)
            throws RecordSetNotFoundException, java.io.IOException {
        ...
    }

    public String[] listRecordSets()
            throws java.io.IOException {
        ...
    }

    ... // métodos e campos não públicos

}
```

2.5 Classe JBase

```
package br.usp.ime.jbase;

public class JBase {

    public static void retrieve(long recordId, DBObject obj)
            throws InvalidRecordIdException, WrongTypeException,
                   java.io.IOException, java.io.UTFDataFormatException {
        ...
    }

    public static void remove(long recordId)
            throws InvalidRecordIdException,
                   java.io.IOException {
        ...
    }

    ... // métodos e campos não públicos

}
```

3 Implementação do JBase

- Algumas analogias ajudam a entender o que deve ser implementado desta etapa do projeto:
  - Um `Database` corresponde a um disco ou partição de disco. Os `RecordSet`s nele contidos correspondem aos arquivos no “disco”.
  - Um `DBFile` também corresponde a um disco ou partição de disco. Qual a diferença entre um `DBFile` e um `Database`? O `DBFile` corresponde a um “disco sem formatação lógica”: ele é uma mera seqüência de blocos, sem nenhuma estrutura adicional. Já o `Database` corresponde a um “disco formatado logicamente”, isto é, um disco no qual existe um sistema de arquivos.

- Um `Database` é implementado sobre um `DBFile`, através de estruturas de gerenciamento de arquivos gravadas no `DBFile`: diretório dos arquivos no `Database`, lista ou mapa de blocos livres, etc. "Formatar logicamente" um disco é criar e inicializar essas estruturas no disco.

- Para simplificar o projeto vamos fixar um "tamanho único" para os blocos. O JBase funcionará com um tamanho de bloco de 4096 bytes. Use este tamanho de bloco tanto para inicializar o `BufferManager` como para criar `DBFile`s.

- Ao contrário dos arquivos do Unix, os `RecordSet`s de um `Database` são organizados em registros de comprimento variável. Nesta etapa do projeto cada `RecordSet` será implementado por um *heap file*[2], a mais simples das estruturas de arquivos orientadas para registros.

- As páginas de um heap file precisam ser formatadas de modo a acomodar registros de comprimento variável. Use o formato de página descrito nas páginas 53 e 54 do texto "Storing Data: Disks and Files", disponível no xerox do CAMAT. Cada heap file deve ser implementado por um diretório de páginas, conforme descrito na página 57 desse texto.

- Para cada `Database` você tem de manter em disco:

  1. uma das duas estruturas (lista ligada ou mapa de bits) para gerenciamento de blocos livres sugeridas na página 43 do texto citado acima;
  2. um diretório dos arquivos (heap files) no `Database`.

- O diretório dos arquivos no `Database` pode ser implementado como um heap file especial, cujos registros tem três campos: o nome do arquivo, a classe dos objetos armazenados no arquivo, e o page id do início do diretório de páginas do arquivo. Defina o "formato lógico" do seu `Database` de modo que seja fácil localizar o diretório de arquivos: faça esse diretório começar sempre na mesma posição do `DBFile`, ou guarde numa posição fixa do `DBFile` o identificador da primeira página do diretório de arquivos.

- Note que `RecordSet` e `DBIterator` não são classes e sim interfaces. Você terá de escrever classes que implementem essas interfaces:

```
class HeapFile implements RecordSet {
    ...
}
class HFIterator implements DBIterator {

    // Iterador retornado por retrieveAll. Recupera todos os registros
    // de um HeapFile.
    ...
}
```

[2] Em portugûes "heap file" seria "arquivo tipo monte". Como o nome sugere, conceitualmente um heap file é um "amontoado de registros", que são armazenados e recuperados sem nenhuma garantia de ordenação. Você não pode assumir que seus registros de `Alunos` serão recuperados em ordem crescente de `numUSP`, por exemplo.

```
class HFExactMatchIterator implements DBIterator {

    // Iterador retornado por um exactMatchQuery. Recupera os registros
    // de um HeapFile que satisfazem a condição do exactMatchQuery.
    ...
}
class HFRangeIterator implements DBIterator {

    // Iterador retornado por um rangeQuery. Recupera os registros
    // de um HeapFile que satisfazem a condição do rangeQuery.
    ...
}
```

- Nesta etapa do projeto os iteradores retornados por `exactMatchQuery` e `rangeQuery` não são nada eficientes: eles examinam todos os registros e "filtram" os que não satisfazem a condição desejada. Você pode implementá-los em termos de um `HFIterator`.

- É muito conveniente ter também uma classe que lide com cada uma das páginas de um `HeapFile` e fique responsável pelas operações "dentro" da página: inserção de registro na página, remoção de registro da página, iteração sobre os registros da página, etc. Você provavelmente achará útil a classe `br.usp.ime.jbase.HFPage`, cujo código fonte está disponível na página do curso. Fique à vontade para usar essa classe do jeito que ela está ou para alterá-la como bem entender.

- Note que nenhum dos métodos da classe `HFPage` foi declarado como `public`. A própria classe foi declarada sem o `public`. Há um motivo para isso: essa classe não deve ser visível para os programas de aplicação, pois ela não faz parte da API do JBase. A classe `HFPage` só deve ser visível para as outras classes do pacote JBase. Tome o mesmo cuidado quando escrever classes que não fazem parte da API do pacote. Revise o código escrito na etapa 1 do projeto tendo em vista a visibilidade das classes e dos métodos.

4 O Teste desta Etapa: uma Aplicação Simples

Como teste desta etapa do projeto escreva uma aplicação simples, que utilize o pacote JBase para armazenar e recuperar objetos. A natureza do programa de aplicação fica a seu critério. Aqui vão algumas sugestões:

- A aplicação mantém registros de alunos, com informações como nome, número USP, endereço, telefone, média nos últimos semestres, etc. O usuário pode cadastrar alunos, alterar os dados de um aluno, remover alunos, procurar alunos por número, nome, etc. Pode também fazer certas consultas: obter os nomes e números dos alunos que tem média entre tal e tal valor, por exemplo.

- A aplicação gerencia o acervo de livros de uma biblioteca. Para cada livro são armazenadas informações como título, autor, editora, ano de edição, quantidade de exemplares que a biblioteca possui, quantidade de exemplares que estão emprestados, etc. O usuário pode cadastrar livros, alterar certos dados de um livro (a quantidade de exemplares, por exemplo), remover livros, procurar livros por autor, título, etc. Pode também fazer certas consultas: obter os títulos e ou autores dos livros publicados entre tal e tal ano, por exemplo.

- A aplicação gerencia o acervo de fitas de uma vídeo-locadora... (Este caso é parecido com o da biblioteca.)
- A aplicação mantém os dados dos funcionários de uma empresa...

O objetivo não é escrever um programa de aplicação completo e realista, mas testar o JBase tanto do ponto de vista do funcionamento ("O meu pacote funciona ou não?") como da conveniência de uso ("É fácil usá-lo?"). Faça sua aplicação explorar ao máximo a API do pacote.

5 O Que Você Deve Entregar

- Pelo sistema de entrega eletrônica de trabalhos: um arquivo `.tar` contendo:
  - todos arquivos fonte (`.java`) desta etapa;
  - um `Makefile` que compila esses arquivos e roda o(s) programa(s) de aplicação.;
  - um arquivo `README` com instruções adicionais que você julgar necessárias. Provavelmente será o caso de incluir instruções de uso do(s) programa(s) de aplicação.

  Os ítens acima devem ser "empacotados" num único arquivo `.tar`, pois o sistema de entrega eletrônica recebe só *um* arquivo. Caso o arquivo `.tar` fique muito grande, comprima-o com o `gzip` e entregue um arquivo `.tar.gz`.
- Em papel: listagens dos arquivos `.java`, do `Makefile` e do arquivo `README`.

Uma pessoa que recebeu o seu arquivo `.tar` (ou `.tar.gz`) e o desempacotou deve poder compilar os fontes e rodar os testes com *muita* facilidade. Use o `Makefile` para automatizar ao máximo esse processo. Caso sejam necessários diretórios organizados de modo especial, inclua tais diretórios no arquivo `.tar`. Se for preciso acertar alguma variável do environment antes de rodar o `make`, não deixe de colocar essa informação no arquivo `README`.

Bom Projeto!