Хранение данных. Сериализация и объектно-реляционные отображения

Алексей Островский

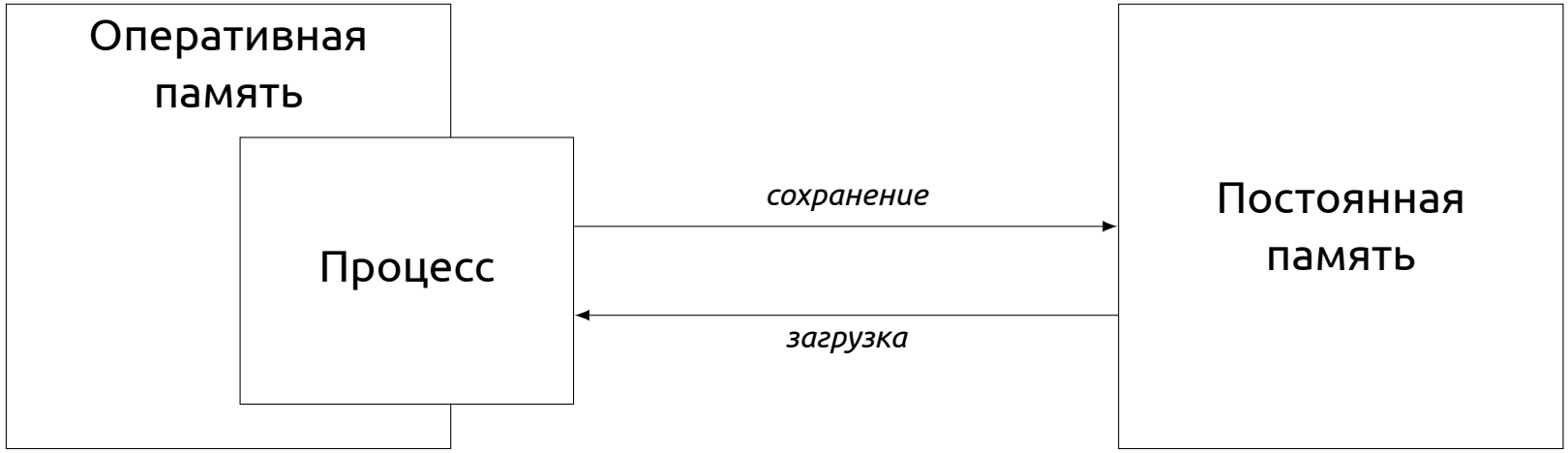
Физико-технический учебно-научный центр НАН Украины

14 мая 2015 г.

Data persistence

Определение

Постоянное хранение данных (англ. *data persistence*) — сохранение информации после прекращения существования процесса, породившего эту информацию.



Data persistence подразумевает двусторонний процесс: загрузку данных из постоянной памяти в оперативную и наоборот

Задача хранения данных

Задача

Добавить к функциональности объекта возможность его хранения в постоянном хранилище данных (напр., в двоичном файле или в реляционной базе данных) при условии минимальной модификации его внутренней структуры.

Проблемы:

- хранение отношений с другими объектами, в т. ч. рекурсивных;
- работа с избыточностью (напр., с множественными ссылками на один объект);
- прозрачное преобразование информации (напр., для минимизации объема хранимых данных);
- независимость от среды выполнения.

Представления данных

Оперативная память:

- формат определяется ABI среды выполнения;
- оптимизация скорости произвольного доступа к данным (напр., выравнивание полей объектов);
- связь с другими объектами через указатели.

Потоки данных (сериализация):

- формат определяется ABI сервиса сериализации или стандартом (JSON, XML, …);
- оптимизация объема данных (напр., преобразование строк UTF-16 $\rightarrow$ UTF-8);
- связь с другими объектами через специальные средства.

Реляционная БД (объектно-реляционные отображения):

- формат определяется схемой БД;
- оптимизация отношений между данными;
- связь с другими объектами через внешние ключи.

Примеры хранения данных

Двоичные потоки данных:

- сохранение / загрузка объектов в виде локальных файлов;
- передача данных в сети, напр., в CORBA.

Текстовые потоки данных:

- передача данных в сети, напр., при работе с веб-сервисами;
- сохранение объектов с произвольной структурой (напр., в БД).

Базы данных:

- хранение данных о моделях в архитектуре Model — View — Controller (MVC);
- хранение сущностей предметной области (англ. *entity*) в Enterprise-приложениях.

Сериализация

Определение

Сериализация (англ. *serialization*), **маршалинг** (англ. *marshalling*) — перевод структур данных или состояния объекта в формат, доступный для хранения или передачи, с возможностью восстановления из сериализованных данных семантически эквивалентного объекта.

Определение

Десериализация (англ. *deserialization, unmarshalling*) — процесс, обратный сериалиазации: восстановление состояния объекта или структуры данных из сериализованного представления.

Сериализация и десериализация могут осуществляться с использованием встроенных или сторонних сервисов

Достоинства и недостатки сериалиазации

Достоинства:

- компактность хранения данных (для двоичных форматов);
- легкость восприятия и отладки (для текстовых форматов);
- универсальность;
- независимость от языка программирования (для текстовых и двоичных форматов, определенных внешними стандартами).

Недостатки:

- линейность формата хранения данных $\Rightarrow$ для извлечения части информации (напр., отдельного поля объекта) в худшем случае требуется прочесть весь поток данных;
- проблемы валидации данных;
- проблемы совместимости для различных сред выполнения.

Классификация методов сериализации

Варианты использования:

- передача данных:
  - в распределенных приложениях (напр., в архитектуре CORBA);
  - при удаленном вызове функций (напр., в веб-сервисах на основе SOAP или REST);
- хранение данных.

Формат данных:

- двоичный — оптимизация объема данных;
- текстовый — читаемость сериализованных данных человеком.

Дополнительные характеристики:

- система типов данных, возможность определения пользовательских типов;
- наличие / отсутствие пользовательских схем данных и возможности валидации;
- поддержка ссылок на части сериализованных данных.

Схема данных

Определение

Схема данных (англ. *data schema*) — спецификация порядка, в котором хранятся сериализованные данные, их семантики, а также описание отображения встроенных и пользовательских структур данных в сериализованный формат и обратно.

Назначение:

- корректная десериализация;
- проверка формата данных, полученных из внешних источников.

Спецификация:

- для двоичных форматов: ABI сервиса сериализации и / или ЯП;
- для текстовых форматов:
  - спецификация формата данных;
  - спецификация формата данных + заданная программистом схема валидации (напр., XML Schema или DTD для XML).

Сериализация в Java

Принципы сериализации:

- Сериализуемые объекты должны помечаться интерфейсом `java.io.Serializable` (не имеет методов).
- Сериализуются все нестатические поля объекта, не отмеченные ключевым словом **`transient`**.
- Корректно сериализуются ссылки на другие объекты (в т. ч. множественные ссылки на один объект и рекурсивные ссылки).
- К сериализуемым классам относятся:
  - примитивные типы (**`byte`**, **`char`**, **`int`**, …);
  - обертки (Byte, Character, Integer, …);
  - массивы;
  - встроенные реализации коллекций (ArrayList, HashSet, HashMap, TreeSet, TreeMap), …

Сериализация в Java

Принципы сериализации (продолжение):

- Для чтения / записи используются потоки ObjectInputStream / ObjectOutputStream.
- Для изменения поведения при чтении / записи используются специальные функции, декларируемые в классе:

```
1 /* Выполняет чтение файла из двоичного объектного потока. */
2 private void readObject(ObjectInputStream in)
3         throws IOException, ClassNotFoundException;
4 /* Выполняет запись объекта в поток. */
5 private void writeObject(ObjectOutputStream out)
6         throws IOException;
```

- Специальные функции вызываются в корректном порядке при наследовании. Сериализуемый класс может быть потомком несериализуемого класса.

Сериализация в Java — пример сериализуемого класса

```
1  public class SerializationTest implements Serializable {
2      /** Идентификатор версии, записываемый при сериализации. */
3      private static final long serialVersionUID = 1L;
4      /** Не сериализуется, т.к. является статическим. */
5      private static final int ANSWER = 42;
6      /** Сериализуемое поле. */
7      private int foo;
8      /** Еще одно сериализуемое поле. */
9      private List<Integer> someList = new ArrayList<Integer>();
10     /** Не сериализуется, т.к. помечено как transient. */
11     private transient Map<Integer, String> cache
12             = new HashMap<Integer, String>();
13     /* Другие поля и методы. */
14
15     private void readObject(ObjectInputStream in) throws IOException {
16         in.defaultReadObject(); // Механизм чтения по умолчанию
17         // При десериализации не вызывается конструктор объекта,
18         // так что несохраненные поля надо инициализировать вручную.
19         this.cache = new HashMap<Integer, String>();
20     }
21 }
```

Сериализация в Java — пример использования

```java
final String filename = "dump";
final SerializationTest testObj = // …

// Запись в поток; основой для потока служит файл
FileOutputStream fos = new FileOutputStream(filename);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(testObj);
oos.close();

// Чтение из потока
FileInputStream fis = new FileInputStream(filename);
ObjectInputStream ois = new ObjectInputStream(ins);
final SerializationTest otherObj = (SerializationTest) ois.readObject();
ois.close();

// При правильной имплементации метода SerializationTest.equals()
// исходный и восстановленный объект должны совпадать
assertEquals(testObj, otherObj);
```

Сериализация в Python

Принципы сериализации:

- Для сериализации используется модуль `pickle` (входит в стандартную библиотеку).
- Для быстрой сериализации может использоваться модуль `cPickle` (аналог pickle, написанный на C).
- Поддерживается сериализация:
  - примитивных типов (**`bool`**, **`int`**, **`long`**, **`float`**, **`complex`**, **`None`**);
  - строк (**`str`**, **`unicode`**);
  - коллекций (**`tuple`**, **`list`**, **`set`**, **`dict`**, ...);
  - функций и классов, определенных на уровне модуля, т. е. не являющихся вложенными (сериализуется только имя класса / функции, но не код или переменные);
  - экземпляров таких классов.
- По умолчанию сериализуется словарь объекта (`obj.__dict__`).
- Для изменения поведения при сериализации используются функции `__getstate__`, `__setstate__`.

Сериализация в Python — пример сериализуемого класса

```
1  class SerializationTest(object):
2      def __init__(self, foo, bar):
3          self.foo = foo
4          self.bar = bar
5
6      def __getstate__(self):
7          odict = self.__dict__.copy() # клонировать словарь объекта
8          del odict['bar'] # не сериализовать поле bar объекта
9          return odict
10
11     def __setstate__(self, state):
12         self.__dict__.update(state) # скопировать все сохраненные поля
13         self.bar = 'bar'
14
15     def __str__(self): return str(self.__dict__)
```

Сериализация в Python — пример использования

```
1  import pickle
2
3  obj = SerializationTest('foo', '')
4  print obj # {'foo': 'foo', 'bar': ''}
5
6  # Запись объекта в файл
7  filename = 'dump'
8  with open(filename, 'wb') as fh:
9      pickle.dump(obj, fh)
10
11 # Чтение объекта из файла
12 with open(filename, 'rb') as fh:
13     other = pickle.load(fh)
14     print type(other) # <class '__main__.SerializationTest'>
15     print other # {'foo': 'foo', 'bar': 'bar'}
```

XML

Определение

XML (extensible markup language) — язык разметки документов, созданный международным комитетом W3C и ориентированный на воспринимаемость человеком и программами.

Преимущества:

- легок для восприятия и отладки человеком;
- широко используется в веб-приложениях (HTML — ≈ частный вариант XML);
- большое количество инструментов для работы с XML во многих ЯП;
- наличие стандартных механизмов валидации (DTD, XML Schema) и преобразования формата данных (XSLT).

Недостатки:

- определенная громоздкость языка;
- отсутствие встроенной семантики типов и структур данных.

Структура XML-документов

Составляющие XML:

- текст;
- теги (открывающиеся `<foo>`, закрывающиеся `</foo>`, пустые `<foo/>`);
- атрибуты тегов (`<foo bar="bazz">`);
- комментарии (`<!-- comment -->`);
- сущности (англ. *entity*) $\simeq$ макросы, использующиеся в тексте для устранения неоднозначностей.

  Пример: `&gt;` заменяет символ `>`, `&lt;` — `<`.
- декларация XML, определяющая версию XML (1.0 или 1.1) и кодировку (UTF-8, UTF-16).

  Пример: `<?xml version="1.1" encoding="UTF-8"?>`.

Пример XML-документа

```xml
<?xml version="1.0" encoding="UTF-8"?>
<reader firstname="Alex" lastname="Ostrovski">
    <location country="Ukraine" city="Kiev" />
    <contacts>
        <email>ostrovski.alex@gmail.com</email>
        <phone type="mobile">000000000</phone>
    </contacts>
    <checkedout>
        <book isbn="0-13-703515-2" language="en">
            <title>Software Engineering, 9th Edition</title>
            <author>Ian Sommerville</author>
            <publisher>Addison-Wesley</publisher>
            <publisher>Pearson Education, Inc.</publisher>
        </book>
        <book isbn="1-59822-033-0" language="en">
            <title>Advanced JavaScript</title>
            <author>Chuck Easttom </author>
            <publisher>Worldware Publishing</publisher>
        </book>
    </checkedout>
</reader>
```

Чтение / запись XML

- **На основе событий.** Отдельные события соответствуют появлению текста, открывающих / закрывающих тегов, атрибутов.

  Использование: для извлечения малого объема информации из больших документов.

  Пример: Simple API for XML (SAX).

- **На основе потоков** чтения / записи (англ. *pull parsing*). Документ представляется как последовательность символов (тегов, атрибутов, текста, …).

  Использование: для извлечения малого объема информации из больших документов.

  Пример: StAX.

Чтение / запись XML (продолжение)

- **Объектный анализатор.** Документ представляется как дерево объектов, соответствующих тегам и другим элементам XML.

  Использование: для анализа документа целиком (напр., в веб-приложениях).

  Пример: Document Object Model (DOM).

- **Привязка данных.** Элементы XML-документа отображаются в поля объектов.

  Использование: для быстрой имплементации сериализации в XML.

  Пример: Java Architecture for XML Binding (JAXB).

Схемы данных в XML

Схема данных в XML:

- определяет семантику элементов (допустимые атрибуты, дочерние элементы, …);
- позволяет проводить проверку корректности (валидацию) XML при чтении.

Определения схемы данных:

- **DTD** (document type definition) — на основе регулярных выражений.

  +: широкая распространенность; краткость по сравнению со схемами на основе XML.

  –: неполная поддержка свойств XML (напр., пространств имен); отсутствие поддержки типов данных.

- **XSD** (XML schema definition) — на основе XML.

  +: поддержка типов данных и сложных ограничений на допустимые элементы и атрибуты; поддержка всех возможностей XML 1.1.

  –: громоздкость.

API для работы с XML

- **XSLT** — декларативный язык программирования на основе XML, позволяющий преобразовывать XML-документы в другие XML-документы.

  Пример использования: преобразование XML, извлеченного из базы данных, в XHTML (вариант HTML, совместимый с XML).

- **XPath** — язык запросов для выбора элементов XML-документа.

  Пример использования: эффективное извлечение информации из XML-документов.

- **XQuery** — функциональный язык программирования для произвольных преобразований XML-документов.

  Пример использования: преобразование XML, извлеченного из базы данных, в HTML или другой формат (напр., JSON).

JSON

Определение

JSON (JavaScript object notation) — формат разметки на основе языка программирования JavaScript, предназначенный для передачи данных в виде пар «атрибут — значение».

Использование:

- в веб-приложениях как альтернатива XML;
- в документно-ориентированных базах данных (напр., MongoDB).

Преимущества:

- легкость восприятия человеком;
- меньший объем по сравнению с XML;
- наличие встроенной семантики типов данных.

Недостатки:

отсутствие стандартных API для многих задач (напр., валидации).

Структура JSON-документов

Типы данных:

- числа (напр., `1`, `-1.5`, `+2.71828E+10`);
- булев тип (**`true`**, **`false`**);
- **`null`**;
- строки (напр., `"a"`, `"string"`);
- массивы — упорядоченная коллекция из нуля или более элементов.

  Примеры: `[]`; `["foo", 5, false]`.
- объекты — неупорядоченный набор строковых ключей (атрибутов) и соответствующих значений.

  Примеры: `{}`; `{"foo": "bar", "bazz": false}`.

Пример документа JSON

```
{
    "firstname": "Alex",
    "lastname": "Ostrovski",
    "location": { "country": "Ukraine", "city": "Kiev" },
    "contacts": {
        "emails": [ "ostrovski.alex@gmail.com" ],
        "phonenumbers": [
            { "type": "mobile", "number": "000000000" }
        ]
    },
    "checkedout": [
        {
            "type": "book",
            "isbn": "0-13-703515-2",
            "language": "en",
            "title": "Software Engineering, 9th Edition",
            "authors": [ "Ian Sommerville" ],
            "publishers": [ "Addison-Wesley", "Pearson Education, Inc." ]
        }
    ]
}
```

API для JSON

- **Чтение / запись:** встроенные или сторонние библиотеки во многих ЯП:
  - JavaScript: JSON (также доступен через библиотеки, напр., jQuery);
  - Java: Google GSON;
  - PHP: функции json_encode, json_decode;
  - Python: модуль json.
- **Валидация:** JSON Schema (на стадии разработки).
- **Удаленный вызов процедур:** JSON-RPC.
- **Загрузка данных из сторонних источников:** JSONP.
- **Преобразование в двоичный формат:** BSON.

Объектно-реляционные отображения

Реляционные БД:

- сущности — строки таблицы или представления в базе данных (т. е. кортежи разнородных данных);
- нормализация с целью сократить объем хранимой информации;
- идентификация сущностей с помощью первичных ключей (primary key);
- связи между сущностями с помощью внешних ключей (foreign key).

ООП:

- сущности — объекты (набор разнородных свойств + методы);
- идентификация сущностей с помощью сравнения;
- связи между сущностями с помощью ссылок на другие объекты.

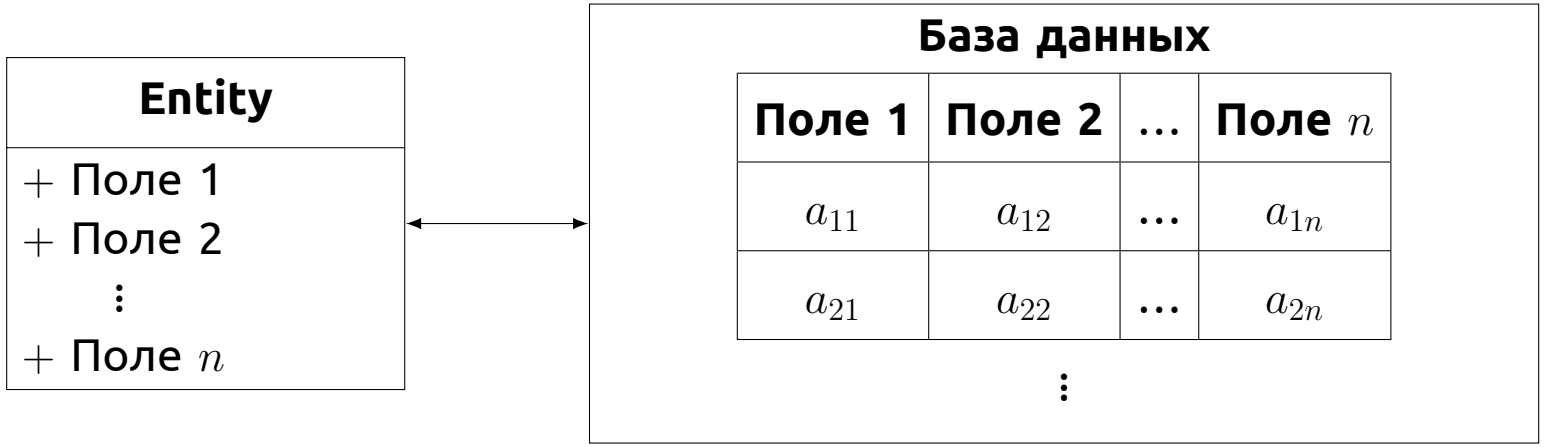
Объектно-реляционные отображения

Задача

Определить двустороннее преобразование от объектно-ориентированного к реляционному представлению данных.

Определение

Объектно-реляционное отображение (англ. *object-relational mapping, ORM*) — технология для взаимодействия с базами данных с использованием парадигмы объектно-ориентированного программирования.



Связь между объектами и СУБД в рамках ORM

Преимущества и недостатки ORM

Преимущества (по сравнению с использованием низкоуровневого доступа к БД):

- высокоуровневые структуры данных (объекты вместо массивов или ассоциативных таблиц) упрощают программирование;
- упрощение управления БД, в частности, миграции.

Преимущества (по сравнению с сериализацией):

- структуризация хранения данных упрощает их анализ;
- встроенные в БД ограничения целостности.

Недостатки:

- повышенное потребление памяти;
- иллюзия простоты: неаккуратное использование ORM может привести к значительному повышению нагрузки на БД.

Базовые понятия ORM

- **Сущность** (англ. *entity*) — ∼ класс в ООП, отдельное понятие предметной области (соответствует таблице или представлению БД). Экземпляры сущности соответствуют строкам таблицы / представления.
- **Первичный ключ** (англ. *primary key*), **идентификатор** — поле или набор полей, позволяющий идентифицировать и различать экземпляры сущности. Соответствует декларации SQL `PRIMARY KEY`.
- **Отношения между сущностями** (англ. *entity relationship*) — связи, определяемые через декларации SQL `FOREIGN KEY`:
  - один к одному;
  - один ко многим;
  - многие ко многим (через вспомогательную таблицу).

Базовые понятия ORM (продолжение)

- **Ленивая загрузка** (англ. *lazy load*) / **немедленная загрузка** (англ. *eager fetching*) — различные стратегии загрузки связанных объектов при извлечении сущности из БД (при загрузке основного объекта с помощью SQL `JOIN` или при попытке доступа).
- **Ограничения** (англ. *constraint*) — условия, накладываемые на значения полей. Соответствуют ограничениям на типы данных SQL (напр., максимальная длина строки для поля типа `VARCHAR(n)`). Ограничения проверяются перед сохранением данных в БД.
- **Язык запросов** (англ. *query language*) — язык, похожий на SQL, предназначенный для выполнения запросов к БД, возвращающих сущности.

Шаблон ActiveRecord

Описание: каждый объект обладает методами для сохранения, обновления и удаления из БД.

Псевдокод (Java):

```
1  // создание объекта
2  Reader reader = new Reader();
3  reader.setFirstName("Alex");
4  reader.save(); // сохранение в БД
5  // модификация объекта
6  // загрузка по первичному ключу
7  Reader reader = Reader.getById(10);
8  reader.setFirstName("Bob");
9  reader.update(); // сохранение в БД
10 // удаление объекта
11 Reader.deleteById(15);
```

Примеры: Yii Framework.

Достоинства: краткость и очевидность кода.

Недостатки: повышенное число запросов к БД.

Шаблон DataMapper

Описание: для извлечения и сохранения объектов в БД используется вспомогательный класс.

Псевдокод (Java):

```
1  Reader reader = new Reader();
2  reader.setFirstName("Alex");
3  entityManager.persist(reader);
4  // загрузка по первичному ключу заданной сущности
5  reader = entityManager.find("Reader", 10);
6  reader.setFirstName("Bob");
7  entityManager.persist(reader);
8  reader = entityManager.find("Reader", 15);
9  entityManager.delete(reader);
10 // сохраняет все изменения в БД
11 entityManager.flush();
```

Примеры: Doctrine (PHP), Java Persistence API.

Достоинства: возможность управления агрегацией операций.

Пример использования

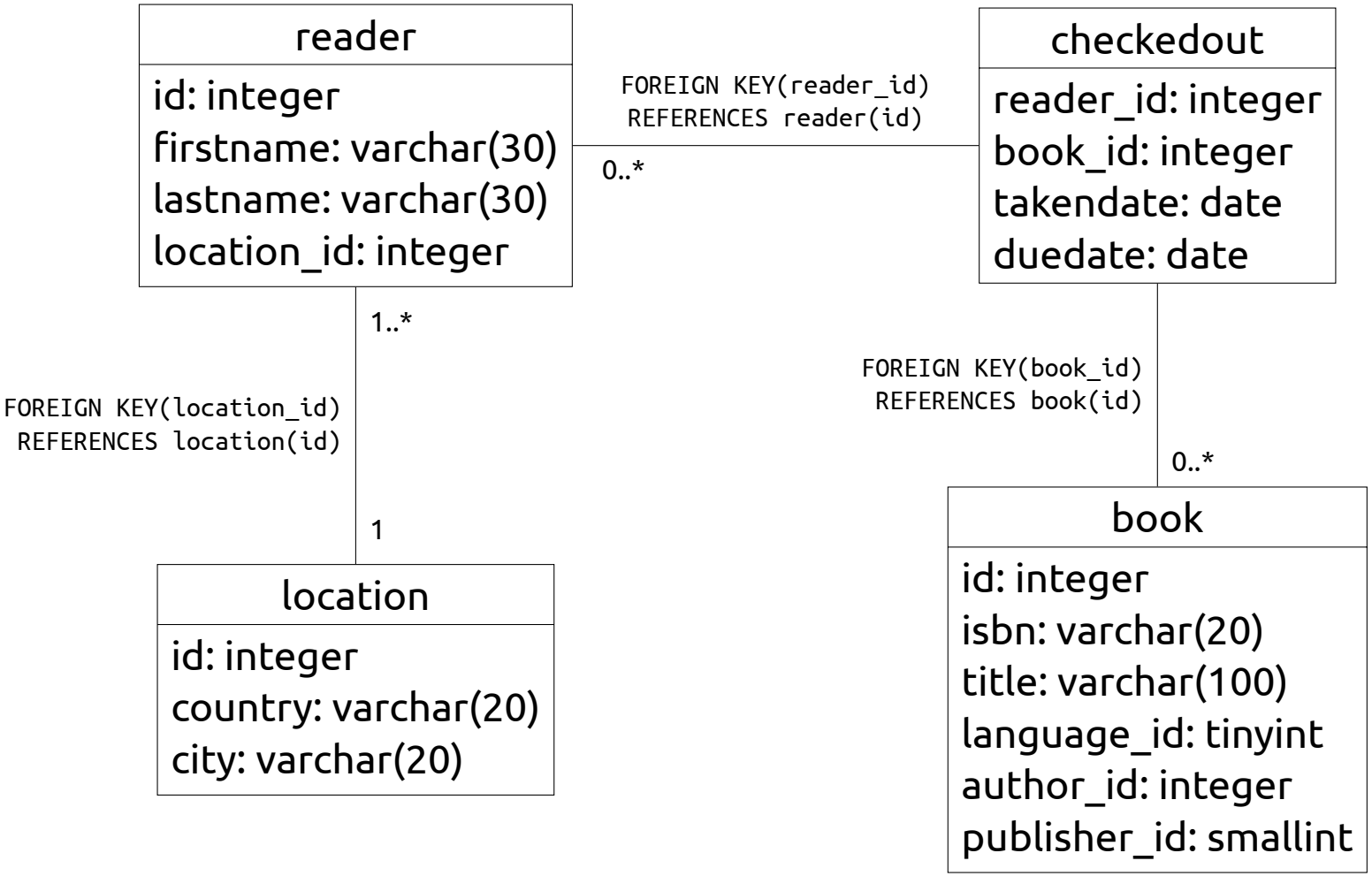


Схема организации данных в БД. Большинство ORM-систем умеют создавать схему на основе классов сущностей.

Пример использования

```java
@Entity
public class Reader implements Serializable {
    /** Идентификатор (генерируется автоматически базой данных). */
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;

    /** Аннотация означает, что при сохранении поле не может равняться null. */
    @NotNull protected String firstName;

    @NotNull protected String lastName;

    /** Отношение «многие (читатели) к одному (местоположению)». */
    @ManyToOne @JoinColumn(name="location_id")
    protected Location location;

    /** Отношение «многие (читатели) ко многим (взятым книгам)». */
    @ManyToMany @JoinTable(name="checkedout")
    protected List<Book> checkedOut;
}
```

Выводы

1. Задача хранения данных (напр., состояния объектов) вне оперативной памяти может решаться с помощью сериализации или создания объектно-реляционного отображения для сущностей предметной области.
2. Сериализация — сохранение данных в поток (двоичный или текстовый). Средства сериализации присутствуют в большинстве языков программирования.
3. Два популярных стандарта текстовой сериализации — XML и JSON — широко используются при передаче данных, напр., в веб-приложениях.
4. Объектно-реляционные отображения используются для установления связи между БД и программой, написанной на объектно-ориентированном ЯП.

Материалы

- Walsh, Norman

  Technical Introduction to XML.

  http://www.xml.com/pub/a/98/10/guide0.html

- ECMA

  Introducing JSON.

  http://www.json.org/

- Oracle

  Introduction to the Java Persistence API.

  http://docs.oracle.com/javaee/7/tutorial/persistence-intro001.htm

  (введение в ORM на примере JPA)

- Doctrine DevTeam

  Getting Started with Doctrine.

  http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/tutorials/getting-started.html

  (введение в ORM на примере Doctrine)

Спасибо за внимание!