> 译自：An Introduction to Hibernate 6
>
> 文中相关链接需要科学上网方可访问，后续有时间再逐个翻译。文章中如果存在任何不准确的地方，欢迎指正。
>
> 尚未完成，不断更新中....

系列文章：

Hibernate 6 中文文档（一）

目录

前言

1. 介绍

1.1. Hibernate 和 JPA

1.2. 使用 Hibernate 编写Java代码

1.3. Hibernate 快速上手

1.4. JPA 快速上手

1.5. 编写持久化模块的代码

1.6. 测试持久化逻辑

1.7. 架构与持久化层

1.8. 概览

前言

Hibernate 6是世界上最受欢迎且功能丰富的对象关系映射（ORM）解决方案的一次重大改版。这次改版几乎触及了Hibernate的每个子系统，包括API、映射注解和查询语言。这个新版本的Hibernate更加强大、更加健壮，而且类型安全性更强。

在这么多改进中，要总结这项工作的重要性非常困难。但以下几个主题最为突出

Hibernate 6：

- 充分利用了过去十年来关系数据库的进展，更新了查询语言以支持现代SQL方言中的众多新构造
- 在不同数据库间表现出更加一致的行为，极大地提高了可移植性，并且从独立于方言的代码中生成更高质量的DDL（数据定义语言）
- 在访问数据库之前更加严格地验证查询，改善了错误报告
- 提高了对象关系映射注解的类型安全性，明确了API、SPI和内部实现的分离，并修复了一些长期存在的架构缺陷
- 移除或废弃了旧版API，为未来的演进奠定了基础
- 更好地使用了Javadoc，为开发人员提供了更多信息

Hibernate 6 和 Hibernate Reactive 现在是Quarkus 3的核心组件，Quarkus 3是Java中最令人激动的云原生开发新环境，而Hibernate依然是几乎每个主要Java框架或服务器的首选持久化解决方案。

不幸的是，Hibernate 6的变化使得大部分现有的关于Hibernate的信息在书籍、博客文章和stackoverflow上都已经过时。

本指南是关于当前特性集和推荐用法的最新、高层次的讨论。它不试图覆盖每个特性，应该与其他文档一起使用：

- Hibernate的详尽Javadoc文档，
- Hibernate查询指南，以及
- Hibernate用户指南。

> Hibernate用户指南详细讨论了Hibernate的大部分方面。但由于要涵盖的信息太多，难以实现可读性，因此它最适合作为参考。在必要时，我们将提供到用户指南相关章节的链接。

1. 介绍

Hibernate通常被视为一个库，它可以被用来轻松地将Java类映射到关系数据库表。但这种看法并没有充分体现关系数据本身的核心作用。因此，关于它的功能更准确的描述应该是：

Hibernate将关系数据以一种自然且类型安全的形式展现给Java程序，使得编写复杂查询和处理查询结果变得容易，让程序能够轻松地将内存中所做的更改与数据库同步，遵循事务的ACID属性，并且在基本持久化逻辑编写后，允许进行性能优化。

这里关注的是关系数据，以及类型安全的重要性。对象/关系映射（ORM）的目标是消除脆弱且不安全的代码，使得长期来看，更容易维护大型程序。

ORM通过解放开发人员手工编写冗长、重复和脆弱的代码，将对象图形转换为数据库表格，以及从扁平的SQL查询结果集重建对象图形的需求，从而减轻了持久性方面的痛点。更妙的是，在基本持久性逻辑编写后，ORM使性能调优变得更加容易。

一个常见的问题是：我应该使用ORM，还是纯SQL？答案通常是：两者都用。JPA和Hibernate是为与手写SQL配合使用而设计的。大多数SQL查询逻辑较好的程序都会受益于ORM的帮助。但是，如果Hibernate在某个特定的SQL查询中上，使查询本身变得更加困难，使用更适合该问题的解决方案才是明智的选择，不需要固执的使用Hibernate！你使用了Hibernate解决持久性方面的问题，并不意味着你必须在所有地方都使用它。

开发人员经常问有关Hibernate和JPA之间关系的问题，因此，首先让我们先来简单了解一下它们的发展史。

1.1. Hibernate 和 JPA

Hibernate 是 Java Persistence API（现在是Jakarta）或JPA背后的灵感来源，并包含了对该规范最新修订版的完整实现。

译者注：

- 例如，旧版本（Java 8以及之前）中，我们通过如下方式引入：

```
import javax.persistence.*;
```

- 现在新版本中（自 Java 9 开始）我们可以使用：

```
import jakarta.persistence.*;
```

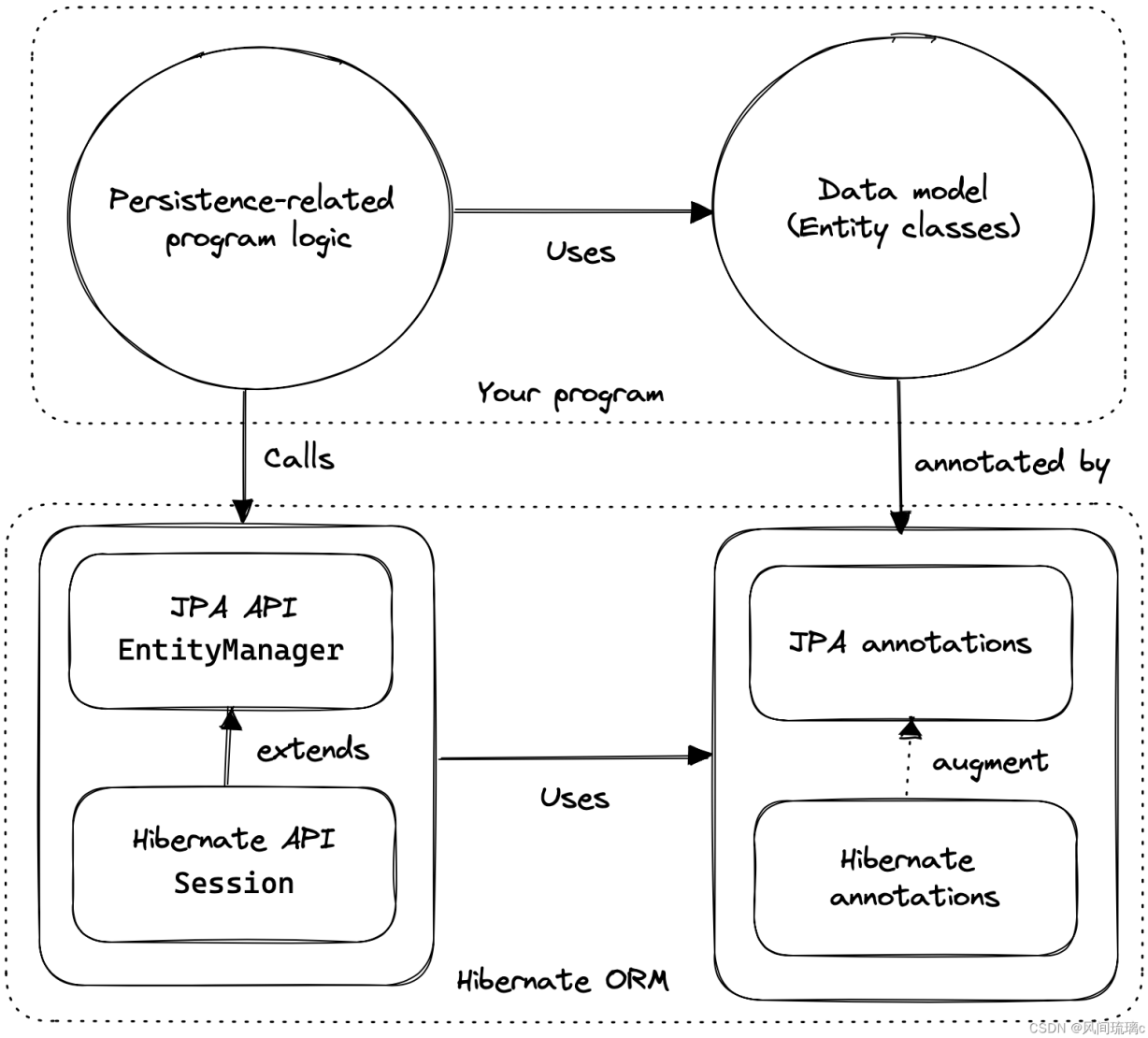
> **Hibernate和JPA的早期历史**
>
> Hibernate项目始于2001年，当时Gavin King对EJB 2中的Entity Beans感到非常沮丧。它很快超越其他开源和商业竞争对手，成为Java中最流行的持久化解决方案，并且与Christian Bauer合著的《Hibernate in Action》成为了一本具有影响力的畅销书。
>
> 在2004年，Gavin和Christian加入了一个名为JBoss的小型初创公司，其他早期的Hibernate贡献者也很快加入进来：Max Rydahl Andersen，Emmanuel Bernard，Steve Ebersole和Sanne Grinovero。
>
> 不久后，Gavin加入了EJB 3专家组，并说服该组废弃Entity Beans，转而采用模仿Hibernate的全新持久性API。后来，TopLink团队的成员也参与其中，Java持久性API成为Sun、JBoss、Oracle和Sybase等主要公司，尤其是在Linda Demichiel的领导下，进行合作演进的产物。
>
> 在接下来的20年中，许多才华横溢的人为Hibernate的发展做出了贡献。我们特别感谢Steve，多年来一直领导着该项目，因为Gavin开始专注于其他工作。

译者注： EJB 2（Enterprise JavaBeans 2）是Java EE（Enterprise Edition）规范中的一个版本，它定义了一种用于构建企业级Java应用程序的组件模型。EJB 2 最初发布于1999年，是EJB 1 的改进版本，它提供了一种分布式计算模型，允许开发者构建分布式、可伸缩和安全的应用程序。

在API方面，我们可以将 Hibernate 的API分为三个基本元素：

- 一组实现了JPA定义API的类，最重要的是EntityManagerFactory和EntityManager接口，以及JPA定义的O/R映射注解。
- 一个本地API，公开了所有可用功能的集合，主要围绕着SessionFactory接口（它扩展了EntityManagerFactory）和Session接口（它扩展了EntityManager）。
- 一组映射注解，这些注解扩充了JPA定义的O/R映射注解，并且可以与JPA定义的接口或本地API一起使用。

Hibernate还为扩展或与Hibernate集成的框架和库提供了一系列SPI（服务提供接口），但我们在这里不关心这些内容。



作为应用程序开发人员，你必须决定是否：

- 使用 Session 和 SessionFactory 编写程序，或
- 在合理的范围内使用 EntityManage r和 EntityManagerFactory 编写代码，以最大程度地提高到其他JPA实现的可移植性，在必要时才回退到本地API。

无论选择哪种路径，你大部分时间都将使用JPA定义的映射注解，并且在处理更高级的映射问题时使用Hibernate定义的注解。

你可能会想知道是否可能仅使用JPA定义的API来开发应用程序，实际上是可能的。JPA是一个非常好的基准，它真正解决了对象/关系映射问题的基本要点。但是，如果没有本地API和扩展映射注解，你将错过Hibernate许多强大的功能。

由于Hibernate存在于JPA之前，并且JPA是以Hibernate为蓝本设计的，我们不幸地在标准API和本地API之间存在一些竞争和名称上的重复。例如：

Hibernate	JPA
org.hibernate.annotations.CascadeType	javax.persistence.CascadeType
org.hibernate.FlushMode	javax.persistence.FlushModeType
org.hibernate.annotations.FetchMode	javax.persistence.FetchType
org.hibernate.query.Query	javax.persistence.Query
org.hibernate.Cache	javax.persistence.Cache
@org.hibernate.annotations.NamedQuery	@javax.persistence.NamedQuery
@org.hibernate.annotations.Cache	@javax.persistence.Cacheable

通常，Hibernate本地API提供了JPA中缺少的一些额外功能，因此这并不算是一个缺点。但这是需要注意的地方。

1.2. 使用 Hibernate 编写Java代码

如果您完全不了解 Hibernate 和 JPA，您可能会好奇，持久化相关功能的代码的层次结构是如何划分的。

通常，我们的持久化相关代码分为两个层次：

1. 一个用Java表示的数据模型，通常以一组带有注解的实体类的形式存在。
2. 一大堆与Hibernate的API交互的函数，用于执行与各种事务相关的持久化操作。

第一点中提到的数据或“领域”模型，通常较容易编写，但做得是否足够好且简洁明了，将会极大地影响到你在第二部分中的成功。

译者注：第一点提到的数据模型，即实体类，通常我们放在entity中，或者model，亦或是 domain 包下。

例如，一个用户实体类：

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Table;
import jakarta.persistence.Transient;
import lombok.*;

import java.util.List;

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class UserData {

    /**
     * 主键
     */
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "c_id")
    private int id;

    /**
     * 登录账号
     */
    @Column(name ="c_username")
    private String username;

    /**
     * 登录密码（加密后）
     */
    @Column(name ="c_password")
    private String password;

    /**
     * 用户昵称
     */
    @Column(name ="c_nickname")
    private String nickname;

    /**
     * 手机号
     */
    @Column(name ="c_telephone")
    private String telephone;

}
```

大多数人将领域模型实现为我们过去称之为“普通Java对象”这一类型。也就是说，这些是简单的Java类，没有直接依赖于技术基础设施，也没有依赖于处理请求、事务管理、通信或与数据库交互的应用程序逻辑。

> 在编写这部分代码时，请多花点时间，尽量生成一个与关系数据模型相近的Java模型。在不真正需要的情况下，避免使用奇异或高级的映射特性。如果有任何疑惑不解的地方，请在外键关系的映射上使用`@ManyToOne`和`@OneToMany(mappedBy=…)`，而不是更复杂的关联映射。

至于上面提到的第二点则要难得多。这部分代码必须：

- 管理事务和会话，
- 通过Hibernate会话与数据库交互，

- 检索并准备UI所需的数据，以及
- 处理失败（如捕获异常）。

> 事务和会话管理的责任，以及从某些类型的失败中恢复的责任，最好由某种框架代码来处理。

译者注：第二点提到的代码一般划分为DAO层，亦或是DaoImpl，DAO接口的实现类。

我们很快将回到这个棘手的问题，讨论这种持久性逻辑应该如何组织，以及它应该如何融入系统的其他部分。

1.3. Hibernate 快速上手

在我们深入了解之前，我们将快速介绍一个基本示例程序，如果你的项目中尚未集成Hibernate，这将帮助你入门。

首先，我们从一个简单的 Gradle 构建文件开始：

译者注：Gradle 和 Maven 都是 Java 项目管理工具，它们用于自动化构建、依赖管理和项目构建的工具。它们的主要目的是简化 Java 项目的构建过程。 Gradle 性能相比 Maven 更加优秀，Spring Boot 也在 2.3.0.M1 版本中对进行了相当重大的更改，开始使用 Gradle 而非 Maven 构建的项目。

build.gradle

```
plugins {
    id 'java'
}

group = 'org.example'
version = '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    // 优秀的ORM
    implementation 'org.hibernate.orm:hibernate-core:6.3.0.Final'

    // Hibernate验证器
    implementation 'org.hibernate.validator:hibernate-validator:8.0.0.Final'
    implementation 'org.glassfish:jakarta.el:4.0.2'

    // Agroal连接池
    implementation 'org.hibernate.orm:hibernate-agroal:6.3.0.Final'
    implementation 'io.agroal:agroal-pool:2.1'

    // 使用Log4j进行日志记录
    implementation 'org.apache.logging.log4j:log4j-core:2.20.0'

    // JPA元模型生成器
    annotationProcessor 'org.hibernate.orm:hibernate-jpamodelgen:6.3.0.Final'

    // HQL的编译时检查
    //implementation 'org.hibernate:query-validator:2.0-SNAPSHOT'
    //annotationProcessor 'org.hibernate:query-validator:2.0-SNAPSHOT'

    // H2数据库
    runtimeOnly 'com.h2database:h2:2.1.214'
}
```

这些依赖项中，只有第一个是运行 Hibernate 所必需的。

接下来，我们添加一个用于Log4j的日志配置文件：

log4j2.properties

```
rootLogger.level = info
rootLogger.appenderRefs = console
rootLogger.appenderRef.console.ref = console

logger.hibernate.name = org.hibernate.SQL
logger.hibernate.level = info

appender.console.name = console
appender.console.type = Console
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %highlight{[%p]} %m%n
```

现在我们需要一些Java代码。我们从实体类开始：

Book.java

```
package org.hibernate.example;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.validation.constraints.NotNull;

@Entity
class Book {
    @Id
    String isbn;

    @NotNull
    String title;

    Book() {}

    Book(String isbn, String title) {
        this.isbn = isbn;
        this.title = title;
    }
}
```

最后，让我们看看配置和实例化 Hibernate 的代码，并要求它进行持久化和查询实体。如果现在这些代码看起来一点都不清晰，不要担心。本文旨在让这些内容变得非常清晰明了。

Main.java

```
package org.hibernate.example;

import org.hibernate.cfg.Configuration;

import static java.lang.Boolean.TRUE;
import static java.lang.System.out;
import static org.hibernate.cfg.AvailableSettings.*;

public class Main {
    public static void main(String[] args) {
        var sessionFactory = new Configuration()
                .addAnnotatedClass(Book.class)
                // 使用H2内存数据库
                .setProperty(URL, "jdbc:h2:mem:db1")
                .setProperty(USER, "sa")
                .setProperty(PASS, "")
                // 使用Agroal连接池
                .setProperty("hibernate.agroal.maxSize", "20")
                // 在控制台显示SQL
                .setProperty(SHOW_SQL, TRUE.toString())
                .setProperty(FORMAT_SQL, TRUE.toString())
                .setProperty(HIGHLIGHT_SQL, TRUE.toString())
                .buildSessionFactory();

        // 导出推断的数据库模式
        sessionFactory.getSchemaManager().exportMappedObjects(true);

        // 持久化一个实体
        sessionFactory.inTransaction(session -> {
            session.persist(new Book("9781932394153", "Hibernate in Action"));
        });

        // 使用HQL查询数据
        sessionFactory.inSession(session -> {
            out.println(session.createSelectionQuery("select isbn||': '||title from Book").getSingleResult());
        });

        // 使用Criteria API查询数据
        sessionFactory.inSession(session -> {
            var builder = sessionFactory.getCriteriaBuilder();
            var query = builder.createQuery(String.class);
            var book = query.from(Book.class);
            query.select(builder.concat(builder.concat(book.get(Book_.isbn), builder.literal(": ")),
```

```
                    book.get(Book_.title)));
            out.println(session.createSelectionQuery(query).getSingleResult());
        });
    }
}
```

在这里，我们使用了 Hibernate 的本地 API 。我们也可以使用 JPA 标准的 API 来达到相同的目的。

1.4. JPA 快速上手

如果我们限制自己只使用 JPA 标准的 API ， 我们需要使用 XML 来配置 Hibernate 。

META-INF/persistence.xml

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="3.0">

    <persistence-unit name="example">

        <class>org.hibernate.example.Book</class>

        <properties>

            <!-- H2内存数据库 -->
            <property name="jakarta.persistence.jdbc.url"
                      value="jdbc:h2:mem:db1"/>

            <!-- 凭据 -->
            <property name="jakarta.persistence.jdbc.user"
                      value="sa"/>
            <property name="jakarta.persistence.jdbc.password"
                      value=""/>

            <!-- Agroal连接池 -->
            <property name="hibernate.agroal.maxSize"
                      value="20"/>

            <!-- 在控制台显示SQL -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>

        </properties>

    </persistence-unit>
</persistence>
```

请注意，我们的 **build.gradle** 和 **log4j2.properties** 文件保持不变。

我们的**实体类**与之前的版本也没有改变。

不幸的是，JPA并没有提供 **inSession()** 方法，所以我们必须自己实现会话和事务管理。我们可以将这些逻辑放入我们自己的 **inSession()** 函数中，这样我们就不必为每个事务重复这些逻辑。再次强调，你现在不需要理解这些代码的所有内容。

Main.java（JPA版本）

```
package org.hibernate.example;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;

import java.util.Map;
import java.util.function.Consumer;

import static jakarta.persistence.Persistence.createEntityManagerFactory;
import static java.lang.System.out;
import static org.hibernate.cfg.AvailableSettings.JAKARTA_HBM2DDL_DATABASE_ACTION;
import static org.hibernate.tool.schema.Action.CREATE;

public class Main {
    public static void main(String[] args) {
```

```
        var factory = createEntityManagerFactory("example",
                // 导出推断的数据库模式
                Map.of(JAKARTA_HBM2DDL_DATABASE_ACTION, CREATE));

        // 持久化一个实体
        inSession(factory, entityManager -> {
            entityManager.persist(new Book("9781932394153", "Hibernate in Action"));
        });

        // 使用HQL查询数据
        inSession(factory, entityManager -> {
            out.println(entityManager.createQuery("select isbn||': '||title from Book").getSingleResult());
        });

        // 使用Criteria API查询数据
        inSession(factory, entityManager -> {
            var builder = factory.getCriteriaBuilder();
            var query = builder.createQuery(String.class);
            var book = query.from(Book.class);
            query.select(builder.concat(builder.concat(book.get(Book_.isbn), builder.literal(": ")),
                    book.get(Book_.title)));
            out.println(entityManager.createQuery(query).getSingleResult());
        });
    }

    // 在一个会话中执行一些操作，进行正确的事务管理
    static void inSession(EntityManagerFactory factory, Consumer<EntityManager> work) {
        var entityManager = factory.createEntityManager();
        var transaction = entityManager.getTransaction();
        try {
            transaction.begin();
            work.accept(entityManager);
            transaction.commit();
        }
        catch (Exception e) {
            if (transaction.isActive()) transaction.rollback();
            throw e;
        }
        finally {
            entityManager.close();
        }
    }
}
```

实际上，我们从不会直接在 **main()** 方法中直接访问数据库。接下来，我们将讨论如何在一个真实系统中组织持久性逻辑。本章的剩余部分不是必须阅读的内容。如果你渴望了解更多关于Hibernate本身的细节，完全可以直接跳到下一章，稍后再回来阅读。

1.5. 编写持久化模块的代码

在一个真实的程序中，像上面展示的代码一样的持久化逻辑通常会与其他类型的代码交织在一起，包括：

- 实施业务域的规则
- 与用户交互的逻辑

因此，许多开发人员通常会迅速——甚至过于迅速，在我们看来——寻求将持久化逻辑隔离到某种独立的架构层中。我们现在要求你暂时压制这种冲动。

译者注：太对了....上面就已经开始忍不住划分dao层，entity层了...

> 使用Hibernate的最简单方法是直接调用 Session 或 EntityManager。如果你是 Hibernate 的新手，使用包装了JPA的框架只会让你的生活变得更加困难。

我们更喜欢一种自底向上的方法来编写我们的代码。我们喜欢思考方法和函数，而不是关于架构层和容器管理的对象。为了说明我们提倡的代码编写方法，让我们考虑一个使用HQL或SQL查询数据库的服务来作为示例。

我们来以下面的代码来举例说明，这是UI和持久化逻辑的混合：

```
@Path("/")
@Produces("application/json")
public class BookResource {
    @GET
    @Path("book/{isbn}")
    public Book getBook(String isbn) {
        var book = sessionFactory.fromTransaction(session -> session.find(Book.class, isbn));
        return book == null ? Response.status(404).build() : book;
    }
}
```

事实上，我们可能也会以类似的方式结束——很难具体指出上面的代码有什么问题，对于如此简单的情况，引入额外的对象可能会使这个代码变得更加复杂。

这段代码非常好的一点是，会话和事务管理是由通用的"框架"代码处理的，就像我们上面推荐的那样。在这种情况下，我们使用了内置在 Hibernate 中的 **fromTransaction()** 方法。但你可能更喜欢使用其他方法，比如：

- 在像 Jakarta EE 或 Quarkus 这样的容器环境中，使用容器管理的事务和容器管理的持久性上下文
- 自己编写

重要的是，**createEntityManager()** 和 **getTransaction().begin()** 之类的调用不属于常规程序逻辑，因为正确处理错误是棘手而乏味的。

现在让我们考虑一个稍微复杂一点的情况。

```
@Path("/")
@Produces("application/json")
public class BookResource {
    private static final int RESULTS_PER_PAGE = 20;

    @GET @Path("books/{titlePattern}/{page:\\d+}")
    public List<Book> findBooks(String titlePattern, int page) {
        var books = sessionFactory.fromTransaction(session -> {
            return session.createSelectionQuery("from Book where title like ?1 order by title", Book.class)
                    .setParameter(1, titlePattern)
                    .setPage(Page.page(RESULTS_PER_PAGE, page))
                    .getResultList();
        });
        return books.isEmpty() ? Response.status(404).build() : books;
    }
}
```

这是可以的，如果你愿意，你甚至可以将代码保持原样。但有一件事情我们或许可以改进。我们喜欢非常短的方法，每个方法只做一件事情，并且看起来有一个机会引入一个。让我们使用我们最喜欢的事情——提取方法重构。我们得到了以下的代码：

```
static List<Book> findBooksByTitleWithPagination(Session session,
                                                 String titlePattern, Page page) {
    return session.createSelectionQuery("from Book where title like ?1 order by title", Book.class)
            .setParameter(1, titlePattern)
            .setPage(page)
            .getResultList();
}
```

这是一个查询方法的例子，一个接受HQL或SQL查询的参数作为参数的函数，并执行查询，将其结果返回给调用者。这就是它的全部功能；它不协调额外的程序逻辑，也不执行事务或会话管理。

最好的方法是使用 **@NamedQuery** 注解指定查询字符串，这样 Hibernate 可以在启动时，也就是创建 **SessionFactory**时，验证查询，而不是在查询第一次执行时。实际上，由于我们在 Gradle 构建（详见1.3. Hibernate快速上手 中 Grable构建）中包含了 Metamodel 生成器（详见 6. Compile-time tooling），查询甚至可以在编译时验证。

我们需要一个地方来放置这个注解，所以让我们将我们的查询方法移动到一个新类中：

```
@CheckHQL // 在编译时验证命名查询
@NamedQuery(name="findBooksByTitle",
            query="from Book where title like :title order by title")
class Queries {

    static List<Book> findBooksByTitleWithPagination(Session session,
                                                     String titlePattern, Page page) {
        return session.createNamedQuery("findBooksByTitle", Book.class)
                .setParameter("title", titlePattern)
                .setPage(page)
                .getResultList();
    }
}
```

请注意，我们的查询方法并没有试图将`EntityManager`隐藏在它的客户端之外。实际上，客户端代码负责向查询方法提供`EntityManager`或`Session`。这是我们整个方法的一个非常独特的特性。

客户端代码可能会：

- 通过调用`inTransaction()`或`fromTransaction()`获得`EntityManager`或`Session`，就像我们上面看到的那样，或
- 在具有容器管理事务的环境中，可以通过依赖注入获得它。

无论哪种情况，协调一个工作单元的代码通常会直接调用Session或EntityManager，如果需要的话，将它传递给辅助方法，比如我们的查询方法。

```
@GET
@Path("books/{titlePattern}")
public List<Book> findBooks(String titlePattern) {
    var books = sessionFactory.fromTransaction(session ->
            Queries.findBooksByTitleWithPagination(session, titlePattern,
                    Page.page(RESULTS_PER_PAGE, page)));
    return books.isEmpty() ? Response.status(404).build
```

你可能会觉得我们的查询方法看起来有点样板化。这是正确的，但我们更关心的是它不够类型安全。实际上，多年来，HQL查询和将参数绑定到查询参数的代码缺乏编译时检查一直是我们对 Hibernate 不满意的主要原因。

幸运的是，现在有了解决这两个问题的方法：在Hibernate 6.3的试验性功能中，我们现在提供了使用元模型生成器为你填充这种查询方法实现的可能性。这个功能是本介绍的一个完整章节的主题，所以现在我们只给你留下一个简单的例子。

假设我们将**Queries**简化为如下所示：

```
interface Queries {
    @HQL("where title like :title order by title")
    List<Book> findBooksByTitleWithPagination(String title, Page page);
}
```

然后，元模型生成器会自动生成一个带有`@HQL`注解方法的实现，它位于一个名为 **Queries_** 的类中。我们可以像调用我们手写的版本一样调用它：

```
@GET
@Path("books/{titlePattern}")
public List<Book> findBooks(String titlePattern) {
    var books = sessionFactory.fromTransaction(session ->
            Queries_.findBooksByTitleWithPagination(session, titlePattern,
                    Page.page(RESULTS_PER_PAGE, page)));
    return books.isEmpty() ? Response.status(404).build() : books;
}
```

在这种情况下，消除的代码量相当小。真正的价值在于提高了类型安全性。现在，在将参数分配给查询参数时，我们会在编译时发现错误。

> 此时，我们相信你对这个想法充满了疑虑。这是完全合理的。我们很愿意在这里回答你的异议，但那会让我们离题太远。所以我们请你暂时搁置这些想法。我们承诺在适当的时候会让它变得合理。并且，在那之后，如果你仍然不喜欢这种方法，请理解它是完全可选的。没有人会来你家强制你接受它。

现在我们大致了解了我们的持久化逻辑可能是什么样子，自然而然地，我们会问如何测试我们的代码。

1.6. 测试持久化逻辑

当我们为持久化逻辑编写测试时，我们需要：

- 一个数据库
- 由我们持久化实体映射的模式的实例
- 一组测试数据，在每个测试开始时处于良好定义的状态。

或许很明显我们应该使用与我们在生产中将要使用的相同的数据库系统进行测试，而且我们肯定应该为这种配置编写一些测试。但另一方面，执行I/O操作的测试比那些不执行I/O操作的测试要慢得多，而且大多数数据库无法配置为在进程内运行。

因此，由于使用Hibernate 6编写的大多数持久化逻辑在不同数据库之间具有极高的可移植性，通常情况下，对于内存中的Java数据库进行测试是有意义的（我们推荐使用H2数据库）。

> 但是，如果我们的持久化代码使用原生SQL，或者使用悲观锁之类的并发管理特性，我们在这里需要小心。

无论我们是针对真实数据库进行测试还是针对内存中的Java数据库进行测试，我们都需要在测试套件开始时导出模式。我们通常在创建Hibernate SessionFactory或JPA EntityManager时执行此操作，因此传统上我们使用了一个配置属性。

JPA标准的属性是 **jakarta.persistence.schema-generation.database.action** 。例如，如果我们使用 **Configuration** 配置 Hibernate，我们可以这样写：

```
configuration.setProperty(AvailableSettings.JAKARTA_HBM2DDL_DATABASE_ACTION,
Action.SPEC_ACTION_DROP_AND_CREATE);
```

或者，在Hibernate 6中，我们可以使用新的SchemaManager API来导出模式，就像我们之前所做的那样。

```
sessionFactory.getSchemaManager().exportMappedObjects(true);
```

由于在许多数据库上执行DDL语句非常慢，我们不希望在每个测试之前都执行这个操作。相反，为了确保每个测试始于具有良好定义状态的测试数据，我们需要在每个测试之前执行两件事：

1. 清理前一个测试留下的任何混乱，然后
2. 重新初始化测试数据。

我们可以使用SchemaManager截断所有表，使数据库模式为空。

```
sessionFactory.getSchemaManager().truncateMappedObjects();
```

在截断表之后，我们可能需要初始化我们的测试数据。我们可以在一个SQL脚本中指定测试数据，例如：

import.sql

```
insert into Books (isbn, title) values ('9781932394153', 'Hibernate in Action');
insert into Books (isbn, title) values ('9781932394887', 'Java Persistence with Hibernate');
insert into Books (isbn, title) values ('9781617290459', 'Java Persistence with Hibernate, Second Edition');
```

如果我们将此文件命名为 **import.sql** ，并将其放置在根类路径中，那就是我们需要做的一切。

否则，我们需要在配置属性 **jakarta.persistence.sql-load-script-source** 中指定该文件。如果我们使用 **Configuration** 配置 Hibernate ，我们可以这样写：

```
configuration.setProperty(AvailableSettings.JAKARTA_HBM2DDL_LOAD_SCRIPT_SOURCE, "/org/example/test-data.sql");
```

> 每次调用 **exportMappedObjects()** 或 **truncateMappedObjects()** 时，都将执行SQL脚本。
>
> 还有另一种混乱可能由测试留下：二级缓存中的缓存数据。我们建议在大多数类型的测试中禁用Hibernate的二级缓存。或者，如果没有禁用二级缓存，那么在每个测试之前我们应该调用：
>
> ```
> sessionFactory.getCache().evictAllRegions();
> ```

现在，假设你遵循了我们的建议，将你的实体和查询方法编写得尽量减少对“基础设施”（即除了JPA和Hibernate之外的库、框架、容器管理的对象甚至是你自己系统的难以从头创建的部分）的依赖。那么，测试持久化逻辑就会变得非常简单！

你需要做的是：

1. 在测试套件的开始时引导 Hibernate 并创建一个 SessionFactory 或 EntityManagerFactory（我们已经知道如何做了）
2. 在每个 **@Test** 方法中创建一个新的 Session 或 EntityManager ，例如使用 **inTransaction()**

> 实际上，有些测试可能需要多个会话。但要小心不要在不同的测试之间泄漏会话。
>
> 我们还需要进行的另一个重要测试是验证我们的O/R映射注解与实际数据库模式的匹配情况。这再次是模式管理工具的职责，要么是：
>
> ```
> configuration.setProperty(AvailableSettings.JAKARTA_HBM2DDL_DATABASE_ACTION, Action.ACTION_VALIDATE);
> ```
>
> 或是
>
> ```
> sessionFactory.getSchemaManager().validateMappedObjects();
> ```
>
> 这个“测试”在很多情况下人们喜欢在生产中启动系统时运行。

1.7. 架构与持久化层

现在让我们考虑一种不同的代码组织方式，这是我们对其持怀疑态度的方式。

在本节中，我们将提供我们的观点。如果你只对事实感兴趣，或者如果你宁愿不阅读可能动摇你目前观点的东西，请随时跳到下一章。

Hibernate是一个与架构无关的库，而不是一个框架，因此与各种Java框架和容器很好地集成。与我们在生态系统中的位置一致，我们历来避免在架构方面提供太多建议。这是一种我们现在可能倾向于后悔的做法，因为由于缺乏建议，结果产生了来自那些提倡架构、设计模式和额外框架的人的建议，我们怀疑这些建议使得使用Hibernate变得不那么愉快。

特别是那些包装JPA的框架似乎增加了冗余代码，同时减少了Hibernate所提供的对数据访问的精细控制。这些框架没有暴露Hibernate的全部功能集，因此程序被迫使用一个功能较弱的抽象。

我们有点怀疑的是，那种古板、教条的传统智慧是：

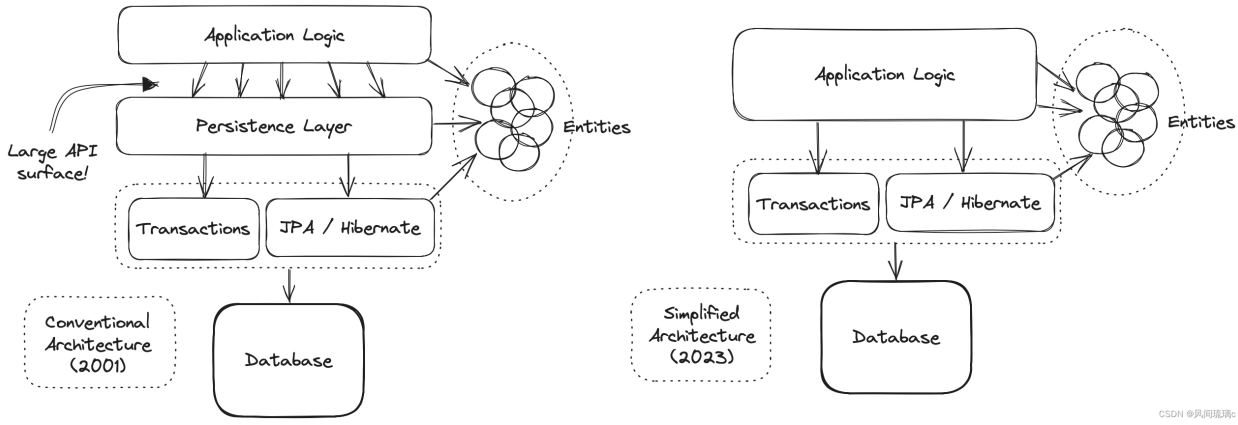
与数据库交互的代码应该位于单独的持久化层。

我们缺乏勇气——甚至可能是信念——坚定地告诉你不要遵循这个建议。但我们请你考虑一下任何架构层的样板代码成本，以及这种成本是否在你的系统环境中真的值得。

为了给这个讨论增加一些背景，尽管有可能我们的介绍在一个很早的阶段就沦为了对这种观点的咆哮，我们要求你在听我们多说一点古老历史的同时，请容忍我们。

> **一个史诗般的DAOs和Repositories故事**
>
> 在Java EE 4的黑暗时代，在Hibernate标准化之前以及JPA在Java企业开发中的日益普及之前，手工编写Hibernate现在负责的那些凌乱的JDBC交互是很常见的。在那个可怕的时期，出现了一个我们过去称之为数据访问对象（DAOs）的模式。DAO给了你一个放置所有那些令人讨厌的JDBC代码的地方，留下了重要的程序逻辑更清晰。
>
> 当Hibernate突然在2001年出现时，开发人员非常喜欢它。但Hibernate没有实现任何规范，许多人希望减少或至少将项目逻辑对Hibernate的依赖局限在某个范围内。一个明显的解决方案是保留DAOs，但是将它们内部的JDBC代码替换为对Hibernate Session的调用。
>
> 我们在这件事情上的角色也是有些责任的。
>
> 在2002年和2003年，这似乎真的是一种相当合理的做法。实际上，我们通过推荐（或者至少不阻止）在《Hibernate in Action》中使用DAOs的方式，为这种方法的流行做出了贡献。我们在这里对这个错误表示歉意，以及对我们花了太长时间才意识到这个错误表示歉意。
>
> 最终，一些人开始相信他们的DAOs使他们的程序免于依赖ORM，使他们能够在需要时用JDBC或其他东西替换掉Hibernate。事实上，这并不是真的——在每次与数据库的交互都是显式和同步的JDBC编程模型与Hibernate中的有状态会话的编程模型之间存在相当深的差异，其中更新是隐式的，SQL语句是异步执行的。
>
> 但是，整个Java中的持久性的风景在2006年4月发生了变化，当时JPA 1.0的最终草案得到了批准。Java现在有了一种标准的ORM方式，具有多个高质量的标准API的实现。DAOs的时代终结了，对吗？
>
> 嗯，不是的。不是的。DAOs被重新命名为“repositories”，并且继续作为连接到JPA的前端而活在世上。但是它们真的发挥了作用吗，还是它们只是多余的额外复杂性和膨胀？它们是一种使堆栈跟踪更难以阅读、代码更难以调试的额外间接层吗？
>
> 我们的观点是，它们主要是多余的。JPA EntityManager就是一个“repository”，它是一个标准的、由整天思考持久性问题的人编写的有明确定义规范的存储库。如果这些存储库框架提供了实际有用的东西——并且不是显然会导致问题的东西——超出了EntityManager提供的功能，我们几十年前就已经将它添加到EntityManager中了。

最终，我们不确定你是否真的需要一个单独的持久化层。至少考虑一下可能直接从你的业务逻辑中调用EntityManager可能是可以接受的。



译者注：很新奇的观点，我已经习惯有DAO层了...

API概览

我们已经听到你在对我们的异端邪说嘶嘶声了。但在你砰然关闭笔记本电脑的盖子，去找蒜和草叉之前，花点时间来权衡一下我们提出的观点。

好吧，如果这让你感觉好一些，将EntityManager视为一个适用于系统中每个实体的单一通用“repository”是一种方式。从这个角度看，JPA就是你的持久化层。将这个抽象包装在一个较不通用的第二个抽象中是否真的有必要呢?

即使一个独立的持久化层是合适的，DAO风格的存储库也不是唯一明确正确的分解方程的方式：

1. 大多数复杂一点的查询涉及多个实体，因此很难确定这样一个查询属于哪个存储库。
2. 大多数查询对特定的程序逻辑片段非常具体，并且在系统中的不同位置之间不会被重用。
3. 一个存储库的各种操作很少相互作用或共享共同的内部实现细节。

事实上，存储库本质上具有非常低的内聚性。如果每个存储库都有多个实现，那么存储库对象的层次可能是有意义的，但实际上几乎没有人这样做。因为它们也与客户端高度耦合，具有非常大的API表面。相反，只有当一个层的API非常窄时，它才容易替换。

> 有些人确实使用模拟存储库进行测试，但我们确实很难看到这种做法有任何价值。如果我们不想对我们的真实数据库运行测试，通常可以通过在内存中的Java数据库（如H2）上运行测试来“模拟”数据库。在Hibernate 6中，这比在旧版本的Hibernate中更好，因为HQL现在在平台之间更具可移植性。

哎呀，让我们继续吧。

1.8. 概览

现在是时候开始我们的旅程，真正理解我们之前看到的代码了。

这个介绍将引导你完成使用Hibernate进行持久化的程序开发中涉及的基本任务：

1. 配置和引导Hibernate，并获取SessionFactory或EntityManagerFactory的实例，
2. 编写领域模型，即一组表示程序中持久类型的实体类，并将其映射到数据库的表，
3. 当模型映射到现有关系模式时，自定义这些映射，
4. 使用Session或EntityManager执行查询数据库并返回实体实例的操作，或者更新数据库中的数据，
5. 使用Hibernate元模型生成器提高编译时类型安全性，
6. 使用Hibernate查询语言（HQL）或本地SQL编写复杂查询，最后
7. 调优数据访问逻辑的性能。

当然，我们将从这个列表的顶部开始，即最不有趣的主题：配置。

Hibernate 6 中文文档（二）

目录

2. 配置和引导

2.1. 将Hibernate包含到项目构建中

2.2. 可选依赖项

2.3. 使用JPA XML进行配置

2.4. 使用Hibernate API进行配置

2.5. 使用Hibernate属性文件进行配置

2.6. 基本配置设置

2.7. 自动模式导出

2.8. 记录生成的SQL语句

2.9. 最小化重复的映射信息

2.10. SQL Server中的国际化字符数据

---

2. 配置和引导

我们希望这一节可以很简短。不幸的是，有几种不同的方法可以配置和引导Hibernate，我们将至少需要详细描述其中两种。

获取Hibernate实例的四种基本方式如下表所示：

使用标准的JPA定义的XML，并使用 **Persistence.createEntityManagerFactory()**	当重要的是在不同JPA实现之间的可移植性时通常选择此选项
使用 `Configuration` 类构建一个 `SessionFactory`	当不需要在不同JPA实现之间进行可移植性时，这个选项更快，提供了一些灵活性并且避免了类型转换
使用 `org.hibernate.boot` 中定义的更复杂的API	主要由框架集成者使用，这个选项超出了本文档的范围
让容器负责引导过程并注入 `SessionFactory` 或 `EntityManagerFactory`	在像WildFly或Quarkus这样的容器环境中使用

这里我们将重点放在前两个选项上。

> 在容器中的Hibernate 实际上，最后一个选项非常受欢迎，因为每个主要的Java应用服务器和微服务框架都内置了对Hibernate的支持。这些容器环境通常还具有自动管理EntityManager或Session的生命周期以及与容器管理的事务的关联的功能。
>
> 要了解如何在这样的容器环境中配置Hibernate，您需要参考所选容器的文档。对于Quarkus，这是相关的文档。

如果您在容器环境之外使用Hibernate，您需要：

- 将Hibernate ORM本身以及适当的JDBC驱动程序作为项目的依赖项引入项目中，以及
- 使用有关您的数据库的信息配置Hibernate，通过指定配置属性。

2.1. 将Hibernate包含到项目构建中

首先，在项目中添加以下依赖项：

```
org.hibernate.orm:hibernate-core:{version}
```

译者注：当然也可以使用maven

```
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>{version}</version>
</dependency>
```

这里的 `{version}` 是您使用的Hibernate版本号。

您还需要为您的数据库添加一个JDBC驱动程序的依赖项。

表2. JDBC驱动程序依赖项

数据库	驱动依赖项
PostgreSQL或CockroachDB	`org.postgresql:postgresql:{version}`
MySQL或TiDB	`com.mysql:mysql-connector-j:{version}`
MariaDB	`org.mariadb.jdbc:mariadb-java-client:{version}`
DB2	`com.ibm.db2:jcc:{version}`
SQL Server	`com.microsoft.sqlserver:mssql-jdbc:${version}`
Oracle	`com.oracle.database.jdbc:ojdbc11:${version}`
H2	`com.h2database:h2:{version}`
HSQLDB	`org.hsqldb:hsqldb:{version}`

这里的 `{version}` 是您的数据库的JDBC驱动程序的最新版本号。

2.2. 可选依赖项

你也可以选择添加以下任意附加功能：

表格 3. 可选依赖项

可选功能	依赖项
SLF4J日志实现	`org.apache.logging.log4j:log4j-core` 或 `org.slf4j:slf4j-jdk14`
JDBC连接池（例如Agroal）	`org.hibernate.orm:hibernate-agroal` 和 `io.agroal:agroal-pool`
Hibernate Metamodel Generator（特别是如果你使用JPA标准的Criteria查询API）	`org.hibernate.orm:hibernate-jpamodelgen`
查询验证器，用于HQL的编译时检查	`org.hibernate:query-validator`
Hibernate Validator，一个Bean验证的实现	`org.hibernate.validator:hibernate-validator` 和 `org.glassfish:jakarta.el`
本地二级缓存支持（通过JCache和EHCache）	`org.hibernate.orm:hibernate-jcache` 和 `org.ehcache:ehcache`
本地二级缓存支持（通过JCache和Caffeine）	`org.hibernate.orm:hibernate-jcache` 和 `com.github.ben-manes.caffeine:jcache`
通过Infinispan实现的分布式二级缓存支持	`org.infinispan:infinispan-hibernate-cache-v60`
用于处理JSON数据类型的JSON序列化库，例如Jackson或Yasson	`com.fasterxml.jackson.core:jackson-databind` 或 `org.eclipse:yasson`
Hibernate Spatial	`org.hibernate.orm:hibernate-spatial`
Envers，用于审计历史数据	`org.hibernate.orm:hibernate-envers`

如果你想使用字段级别的延迟加载，还可以将Hibernate字节码增强器添加到你的Gradle构建中。

2.3. 使用JPA XML进行配置

在遵循JPA标准的方法中，我们会提供一个名为 `persistence.xml` 的文件，通常放在持久化归档（即包含我们的实体类的.jar文件或目录）的 `META-INF` 目录下。

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="2.0">

    <persistence-unit name="org.hibernate.example">

        <class>org.hibernate.example.Book</class>
        <class>org.hibernate.example.Author</class>

        <properties>
            <!-- PostgreSQL -->
            <property name="jakarta.persistence.jdbc.url"
                      value="jdbc:postgresql://localhost/example"/>

            <!-- Credentials -->
            <property name="jakarta.persistence.jdbc.user"
                      value="gavin"/>
            <property name="jakarta.persistence.jdbc.password"
                      value="hibernate"/>

            <!-- Automatic schema export -->
            <property name="jakarta.persistence.schema-generation.database.action"
                      value="drop-and-create"/>

            <!-- SQL statement logging -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>

        </properties>

    </persistence-unit>
```

```
</persistence>
```

`<persistence-unit>` 元素定义了一个命名持久化单元，其中包括：

- 一组相关联的实体类型，以及
- 一组默认的配置设置，可以在运行时进行增加或覆盖。

每个 `<class>` 元素指定了一个实体类的完全限定名。

> **扫描实体类**
>
> 在一些容器环境中，例如任何Java EE容器，`<class>` 元素是不必要的，因为容器将会扫描归档文件以查找被 `@Entity` 注解标记的类，并自动识别这些类。

每个 `<property>` 元素指定了一个配置属性及其值。请注意：

- **jakarta.persistence** 命名空间中的配置属性是JPA规范定义的标准属性。
- **hibernate** 命名空间中的属性是特定于Hibernate的属性。

我们可以通过调用 **Persistence.createEntityManagerFactory()** 方法来获得一个 **EntityManagerFactory**：

```
EntityManagerFactory entityManagerFactory =
 Persistence.createEntityManagerFactory("org.hibernate.example");
```

如果需要，我们可以覆盖 `persistence.xml` 中指定的配置属性：

```
EntityManagerFactory entityManagerFactory =    Persistence.createEntityManagerFactory("org.hibernate.example",
          Map.of(AvailableSettings.JAKARTA_JDBC_PASSWORD, password));
```

2.4. 使用Hibernate API进行配置

或者，古老的 `Configuration` 类允许在Java代码中配置Hibernate的实例。

```
SessionFactory sessionFactory =
        new Configuration()
            .addAnnotatedClass(Book.class)
            .addAnnotatedClass(Author.class)
            // PostgreSQL
            .setProperty(AvailableSettings.JAKARTA_JDBC_URL, "jdbc:postgresql://localhost/example")
            // Credentials
            .setProperty(AvailableSettings.JAKARTA_JDBC_USER, user)
            .setProperty(AvailableSettings.JAKARTA_JDBC_PASSWORD, password)
            // Automatic schema export
            .setProperty(AvailableSettings.JAKARTA_HBM2DDL_DATABASE_ACTION,
                         Action.SPEC_ACTION_DROP_AND_CREATE)
            // SQL statement logging
            .setProperty(AvailableSettings.SHOW_SQL, TRUE.toString())
            .setProperty(AvailableSettings.FORMAT_SQL, TRUE.toString())
            .setProperty(AvailableSettings.HIGHLIGHT_SQL, TRUE.toString())
            // Create a new SessionFactory
            .buildSessionFactory();
```

`Configuration` 类从Hibernate的非常早期（1.0版本之前）就一直存在，所以看起来并不特别现代。另一方面，它非常易于使用，并且暴露了一些 `persistence.xml` 不支持的选项。

高级配置选项 实际上，`Configuration` 类只是 `org.hibernate.boot` 包中定义的更现代、更强大但更复杂的API的一个简单外观。如果你有非常高级的需求，例如，如果你正在编写一个框架或实现一个容器，这个API是非常有用的。你可以在用户指南和 `org.hibernate.boot` 包级别的文档中找到更多信息。

2.5. 使用Hibernate属性文件进行配置

如果我们使用Hibernate配置API，但不想将某些配置属性直接放在Java代码中，我们可以在一个名为 `hibernate.properties` 的文件中指定它们，然后将文件放在根类路径下。

```
# PostgreSQL
jakarta.persistence.jdbc.url=jdbc:postgresql://localhost/example
# Credentials
jakarta.persistence.jdbc.user=hibernate
jakarta.persistence.jdbc.password=zAh7mY$2MNshzAQ5

# SQL statement logging
hibernate.show_sql=true
hibernate.format_sql=true
hibernate.highlight_sql=true
```

在这个属性文件中，我们可以将一些配置属性（如数据库URL、用户名、密码等）放置在文件中，而不是硬编码到Java代码中。Hibernate会在类路径下查找这个文件，并使用其中的属性进行配置。

2.6. 基本配置设置

`AvailableSettings`类枚举了Hibernate所理解的所有配置属性。

当然，我们不会在本章节中覆盖每一个有用的配置设置。相反，我们将提到你需要开始的那些，以及一些其他重要的设置，尤其是在我们谈到性能调优时。

> Hibernate有很多开关和切换。请不要乱用这些设置；它们中的大多数很少需要使用，而且许多只是为了与Hibernate的旧版本保持向后兼容。几乎所有这些设置的默认行为都是我们建议的行为，经过精心选择。

你真正需要开始的属性有以下三个：

表4. JDBC连接设置

配置属性名	目的
jakarta.persistence.jdbc.url	数据库的JDBC URL
jakarta.persistence.jdbc.user	数据库的用户名
jakarta.persistence.jdbc.password	数据库的密码

> 在Hibernate 6中，你不需要指定 hibernate.dialect 。Hibernate SQL方言将自动为你确定。只有在你使用自定义的用户编写的Dialect类时才需要指定此属性。
>
> 类似地，在使用支持的数据库之一时，也不需要 hibernate.connection.driver_class 或jakarta.persistence.jdbc.driver

连接池化JDBC连接是一个非常重要的性能优化。你可以使用以下属性设置Hibernate内置连接池的大小：

表5. 内置连接池大小

配置属性名	目的
hibernate.connection.pool_size	内置连接池的大小

> 默认情况下，Hibernate使用一个简单的内置连接池。这个连接池不适合在生产环境中使用，稍后，当我们讨论性能时，我们将看到如何选择一个更强大的实现。

另外，在容器环境中，你至少需要以下其中一个属性：

表6. 事务管理设置

配置属性名	目的
jakarta.persistence.transactionType	（可选，默认为JTA）确定事务管理是通过JTA还是资源本地事务。如果不使用JTA，指定RESOURCE_LOCAL。
jakarta.persistence.jtaDataSource	JTA数据源的JNDI名称
jakarta.persistence.nonJtaDataSource	非JTA数据源的JNDI名称

在这种情况下，Hibernate将从一个受容器管理的数据源中获取池化的JDBC数据库连接。

2.7. 自动模式导出

你可以让Hibernate从你在Java代码中指定的映射注解中推断出数据库模式，并在初始化时导出该模式，方法是指定以下一个或多个配置属性：

表7. 模式管理设置

配置属性名	目的
jakarta.persistence.schema-generation.database.action	如果是drop-and-create，则首先删除模式，然后导出表、序列和约束。 如果是create，则导出表、序列和约束，而不尝试首先删除它们。 如果是create-drop，则在SessionFactory启动时删除模式并重新创建它，此外，在SessionFactory关闭时删除模式。 如果是drop，则在SessionFactory关闭时删除模式。 如果是validate，则验证数据库模式而不进行更改。 如果是update，则只导出模式中缺少的部分。
jakarta.persistence.create-database-schemas	（可选）如果为true，则自动创建模式和目录
jakarta.persistence.schema-generation.create-source	（可选）如果是metadata-then-script或script-then-metadata，则在导出表和序列时执行额外的SQL脚本
jakarta.persistence.schema-generation.create-script-source	（可选）要执行的SQL DDL脚本的名称
jakarta.persistence.sql-load-script-source	（可选）要执行的SQL DML脚本的名称

这个特性在测试中非常有用。

> 预初始化数据库中的测试数据或“参考”数据的最简单方法是将一个SQL插入语句列表放入一个名为`import.sql`的文件中，并使用属性`jakarta.persistence.sql-load-script-source`指定该文件的路径。我们已经看到了这种方法的一个例子，它比编写Java代码实例化实体实例并在每个实例上调用`persist()`方法更清晰。

正如我们前面提到的，通过编程方式控制模式导出也是有用的。

`SchemaManager` API允许对模式导出进行编程控制：

```
sessionFactory.getSchemaManager().exportMappedObjects(true);
```

JPA具有更有限和不太符合人体工程学的API：

```
Persistence.generateSchema("org.hibernate.example", Map.of(JAKARTA_HBM2DDL_DATABASE_ACTION, CREATE))
```

2.8. 记录生成的SQL语句

要查看发送到数据库的生成SQL，你有两种选择。

一种方法是将属性`hibernate.show_sql`设置为`true`，Hibernate将直接将SQL记录到控制台。通过启用格式化或突出显示，你可以使输出更易读。这些设置在故障排除生成的SQL语句时非常有帮助。

表8. 将SQL记录到控制台的设置

配置属性名	目的
hibernate.show_sql	如果为true，则将SQL直接记录到控制台
hibernate.format_sql	如果为true，则以多行、缩进格式记录SQL
hibernate.highlight_sql	如果为true，则通过ANSI转义码的语法突出显示记录SQL

或者，你可以使用你首选的SLF4J日志实现启用类别`org.hibernate.SQL`的调试级别日志记录。

例如，如果你使用的是Log4J 2（如上所述的Optional dependencies），在你的`log4j2.properties`文件中添加以下行：

```
# SQL执行
logger.hibernate.name = org.hibernate.SQL
logger.hibernate.level = debug

# JDBC参数绑定
logger.jdbc-bind.name=org.hibernate.orm.jdbc.bind
logger.jdbc-bind.level=trace
# JDBC结果集提取
logger.jdbc-extract.name=org.hibernate.orm.jdbc.extract
logger.jdbc-extract.level=trace
```

但是使用这种方法我们无法得到漂亮的语法突出显示。

2.9. 最小化重复的映射信息

以下属性非常有用，可以最小化你需要在@Entity注解的@Table和@Column注解中显式指定的信息量，我们将在下面的对象/关系映射中讨论这个问题：

表9. 最小化显式映射信息的设置

配置属性名	目的
hibernate.default_schema	未明确声明模式的实体的默认模式名称
hibernate.default_catalog	未明确声明目录的实体的默认目录名称
hibernate.physical_naming_strategy	实现你的数据库命名标准的PhysicalNamingStrategy
hibernate.implicit_naming_strategy	当在注解中没有指定名称时，指定如何推断关系对象的“逻辑”名称的ImplicitNamingStrategy

> 编写你自己的 `PhysicalNamingStrategy` 和/或 `ImplicitNamingStrategy` 是减少实体类上注解混乱的一种特别好的方式，并且可以实现你的数据库命名约定，因此我们认为你应该在任何非平凡数据模型中这样做。我们将在命名策略中详细介绍它们。

2.10. SQL Server中的国际化字符数据

默认情况下，SQL Server的 **char** 和 **varchar** 类型不支持Unicode数据。但是Java字符串可以包含任何Unicode字符。因此，如果你在使用SQL Server，可能需要强制Hibernate使用 **nchar** 和 **nvarchar** 列类型。

表10. 使用国际化字符数据的设置

配置属性名	目的
hibernate.use_nationalized_character_data	使用 **nchar** 和 **nvarchar** 而不是 **char** 和 **varchar**

另一方面，如果只有一些列存储国际化数据，请使用@Nationalized注解指示实体的字段，这些字段映射这些列。

> 或者，你可以配置SQL Server使用启用UTF-8的校对_UTF8。

Hibernate 6 中文文档（三）

目录

3. 实体

3.1. 实体类

3.2. 访问类型

3.3. 实体类继承

3.4. 标识属性（主键）

3.5. Hibernate生成的标识符

3.6. 自然键作为标识符

3.7. 复合标识符

3.8. 版本属性

3.9. 自然标识符属性

3.10. 基本属性

3.11. 枚举类型

3.12. 转换器

3.13. 组合基本类型

3.14. 可嵌入对象

3.16. 多对一

3.17. 一对一关联（第一种方式）

3.18. 一对一关联（第二种方式）

3.19. 多对多关联

3.20. 基本值和可嵌入对象的集合

3.21. 映射到 SQL 数组的集合

3.22. 映射到单独表的集合

3.23. 注解总结

3.24. equals() 和 hashCode()

3. 实体

实体指的是一个Java类，它表示关系数据库表中的数据。我们说该实体进行了映射或映射到表。少数情况下，一个实体可能会聚合来自多个表的数据，但我们稍后会讨论这一点。

一个实体一般有多个属性，这些属性（也称作字段），

译者注：英文中一般用*attributes*、properties、fields来指代）

映射到表的列。特别是，每个实体必须有一个标识符 或 ID，它映射到表的主键。这个 ID 允许我们将表的一行数据与Java类的一个实例唯一关联起来，至少在给定的持久性上下文中是这样的。

我们将在后面探讨持久性上下文的概念。现在，将其视为 ID 和实体实例之间的一对一映射。

Java类的实例不能超出其所属的虚拟机。但是，我们可以认为一个实体实例具有超越特定内存实例的生命周期。通过将其 ID 提供给 Hibernate，我们可以在新的持久性上下文中重新创建该实例，只要关联的行存在于数据库中。因此，操作 **persist()** 和 **remove()** 可以被认为标志着实体的生命周期的开始和结束，至少在持久性方面是这样的。

因此，ID 表示实体的持久标识，这个标识超越了内存中的特定实例。这是实体类本身和其属性值之间的一个重要区别——实体具有持久标识，并且在持久性方面具有明确定义的生命周期，而表示其属性值之一的 String 或 List 则没有。

通常，一个实体与其他实体存在关联。通常，两个实体之间的关联映射到一个数据库表中的外键。一组相互关联的实体通常被称为领域模型，尽管数据模型也是一个非常好的术语。

3.1. 实体类

一个实体类必须：

- 是一个非final类
- 具有一个无参数的非私有构造函数

另一方面，实体类可以是具体类（即可实例化的类）或抽象类，并且可以有任意数量的其他构造函数。

实体类可以是一个静态内部类。

每个实体类必须用 **@Entity** 进行注解。

```
@Entity
class Book {
    Book() {}
    // ...
}
```

或者，可以通过为该类提供一个基于XML的映射来将其标识为实体类型。

使用XML进行实体映射

在使用基于XML的映射时，可以使用 元素来声明一个实体类：

```
<entity-mappings>
    <package>org.hibernate.example</package>

    <entity class="Book">
        <attributes> ... </attributes>
    </entity>

    ...
</entity-mappings>
```

由于JPA规范定义的 orm.xm l映射文件格式与基于注解的映射紧密相关，因此在这两种选项之间进行转换通常很容易。

在这个简介中，我们不会对基于XML的映射做更多的介绍，因为这不是我们首选的做事方式。

> **“动态”模型**
>
> 我们喜欢将实体表示为类，因为这些类为我们提供了数据的类型安全模型。但是Hibernate还有能力将实体表示为java.util.Map的无类型实例。如果你感兴趣，可以在用户指南中找到更多信息。

> 对于一个非常特定的通用代码来说，这听起来可能是一个奇怪的功能。例如，Hibernate Envers是Hibernate实体的一个很好的审计/版本控制系统。Envers使用map来表示其数据的版本化模型。

3.2. 访问类型

每个实体类都有一个默认的访问类型，可以是：

- 直接字段访问（field access）
- 属性访问（property access）

Hibernate会根据属性级别的注解的位置自动确定访问类型。具体来说：

- 如果一个字段被标记为 **@Id**，将使用字段访问。
- 如果一个getter方法被标记为 **@Id**，将使用属性访问。

在Hibernate刚开始的时候，属性访问在Hibernate社区中非常受欢迎。然而，在今天，字段访问更为普遍。

> 默认的访问类型可以使用@Access注解来明确指定，但我们强烈不推荐这样做，因为这样做既难看又没必要。

> 映射的注解应该放置得一致：
>
> - 如果@Id注解了一个字段，其他映射的注解也应该放在字段上。
> - 如果@Id注解了一个getter方法，其他映射的注解应该放在getter方法上。

理论上，可以使用属性级别的@Access注解来混合字段和属性访问。但我们不建议这样做。

像Book这样的实体类，它没有扩展任何其他实体类，被称为根实体。每个根实体必须声明一个标识属性。

译者注：两者的区别举例如下：
①这是使用lombok简化getset方法，将注解直接加到实体类的字段上的写法：

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "t_user_data")
public class UserData extends BaseEntity {

    /**
     * 主键 ID
     */
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "c_id")
    private int id;

    /**
     * 登录账号
     */
    @Column(name ="c_username")
    private String username;

    /**
     * 登录密码（加密后）
     */
    @Column(name ="c_password")
    private String password;

    /**
     * 用户昵称
     */
    @Column(name ="c_nickname")
    private String nickname;
}
```

②这是较早的时候，用IDE自带的功能生成getset方法，然后将注解加在getset方法上的写法：

```
package com.codawave.pve.model.main;


import com.codawave.pve.model.base.BaseEntity;
import com.codawave.pve.model.shiroAccess.Roles;
import jakarta.persistence.*;
import lombok.*;
```

```java
import java.util.List;


@Entity
@Table(name = "t_user_data")
public class UserData {
    /**
     * 主键 ID
     */
    private int id;

    /**
     * 登录账号
     */
    private String username;

    /**
     * 登录密码（加密后）
     */
    private String password;

    /**
     * 用户昵称
     */
    private String nickname;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "c_id")
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
    @Column(name = "c_username")
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
    @Column(name = "c_password")
    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
    @Column(name = "c_nickname")
    public String getNickname() {
        return nickname;
    }

    public void setNickname(String nickname) {
        this.nickname = nickname;
    }

}
```

相比第一种写法， 第二种写法既冗余还不美观，因此现在更普遍使用第一种写法。

3.3. 实体类继承

一个实体类可以继承另一个实体类。

```
@Entity
class AudioBook extends Book {
    AudioBook() {}
    ...
}
```

子类实体继承了它所扩展的每个实体的所有持久化属性。

根实体（Root Entity Class，指的是没有继承其他实体类的实体类，继承层次中的最上层）还可以扩展另一个类，并从另一个类继承映射的属性。但在这种情况下，声明映射属性的类必须被标记为 **@MappedSuperclass**。

```
@MappedSuperclass
class Versioned {
    ...
}

@Entity
class Book extends Versioned {
    ...
}
```

根实体类必须声明一个带有 **@Id** 注解的属性，或者从 **@MappedSuperclass** 中继承一个。子类实体始终继承根实体的标识属性。它不能声明自己的@Id属性。

3.4. 标识属性（主键）

标识属性通常是一个字段：

```
@Entity
class Book {
    Book() {}

    @Id
    Long id;

    ...
}
```

但它也可以是一个属性：

```
@Entity
class Book {
    Book() {}

    private Long id;

    @Id
    Long getId() { return id; }
    void setId(Long id) { this.id = id; }

    ...
}
```

标识属性必须被注解为 **@Id** 或 **@EmbeddedId** 。

标识值可以通过以下两种方式之一分配：

1. **由应用程序分配**，也就是由你的Java代码分配。
2. **由Hibernate生成并分配。**

我们将首先讨论第二种选项。

3.5. Hibernate生成的标识符

标识符通常是系统生成的，这种情况下应该使用 @GeneratedValue 注解：

```
@Id @GeneratedValue
Long id;
```

> 系统生成的标识符，或者称为代理键，使得演变或重构关系数据模型变得更加容易。如果你有自由定义关系模式的权限，我们建议使用代理键。另一方面，如果你像大多数情况下一样，正在使用一个预先存在的数据库模式，你可能没有这个选项。

JPA 定义了以下生成 id 的策略，它们由 GenerationType 枚举表示：

表11. 标准 id 生成策略

策略	Java 类型	实现方式
GenerationType.UUID	UUID 或 String	Java UUID
GenerationType.IDENTITY	Long 或 Integer	标识符或自增列
GenerationType.SEQUENCE	Long 或 Integer	数据库序列
GenerationType.TABLE	Long 或 Integer	数据库表
GenerationType.AUTO	Long 或 Integer	根据数据库标识符类型和能力选择 SEQUENCE、TABLE 或 UUID

例如，以下 UUID 是通过 Java 代码生成的：

```
@Id @GeneratedValue UUID id;  // AUTO 策略基于字段类型选择 UUID
```

这个 id 对应于 SQL 的标识符、自动增长列或 bigserial 列：

```
@Id @GeneratedValue(strategy=IDENTITY) Long id;
```

`@SequenceGenerator` 和 `@TableGenerator` 注解允许进一步控制 SEQUENCE 和 TABLE 的生成方式。考虑下面的序列生成器：

```
@SequenceGenerator(name="bookSeq", sequenceName="seq_book", initialValue = 5, allocationSize=10)
```

使用以下定义的数据库序列生成值：

```
create sequence seq_book start with 5 increment by 10
```

请注意，Hibernate 不必每次需要新标识符时都访问数据库。相反，给定的过程获得一个大小为 allocationSize 的 id 块，并且只需要在块用尽时才需要访问数据库。当然，缺点是生成的标识符不是连续的。

> 如果你让 Hibernate 导出数据库模式，序列的定义将具有正确的 start with 和 increment 值。但是如果你正在使用Hibernate之外管理的数据库模式，请确保 **@SequenceGenerator** 的 initialValue 和 allocationSize 成员与 DDL 中指定的 start with 和 increment 匹配。

现在，任何标识符属性都可以使用名为 `bookSeq` 的生成器：

```
@Id
@GeneratedValue(strategy=SEQUENCE, generator="bookSeq")  // 引用在别处定义的生成器
Long id;
```

实际上，将 `@SequenceGenerator` 注解放在使用它的 `@Id` 属性上是非常常见的：

```
@Id
@GeneratedValue(strategy=SEQUENCE, generator="bookSeq")  // 引用下面定义的生成器
@SequenceGenerator(name="bookSeq", sequenceName="seq_book", initialValue = 5, allocationSize=10)
Long id;
```

> JPA 的 id 生成器可以在实体之间共享。**@SequenceGenerator** 或 **@TableGenerator** 必须有一个名字，并且可以在多个 id 属性之间共享。这与将使用生成器的 `@Id` 属性进行注解的常见做法有些不协调！

正如你所见，JPA 为系统生成的 id 提供了相当充分的支持。然而，这些注解本身比它们应该的更加侵入式，而且没有明确定义的方法来扩展此框架以支持自定义 id 生成策略。**@GeneratedValue** 不能用于未标注 **@Id** 的属性。由于自定义 id 生成是一个相当常见的需求，Hibernate 提供了一个非常精心设计的用户定义生成器框架，我们将在用户定义生成器中进行讨论。

3.6. 自然键作为标识符

并非每个标识符属性都映射到（系统生成的）代理键。对于系统用户具有意义的主键称为自然键。

当表的主键是自然键时，我们不使用 @GeneratedValue 注解标记标识符属性，将为标识符属性分配一个值是应用程序代码的责任。

```
@Entity
class Book {
    @Id
    String isbn;

    ...
}
```

特别感兴趣的是由多个数据库列组成的自然键，这样的自然键被称为复合键。

3.7. 复合标识符

如果你的数据库使用复合键，你将需要多个标识符属性。在JPA中，有两种映射复合键的方式：

1. 使用 @IdClass
2. 使用 @EmbeddedId

在实体类中，最直观的方式是使用多个带有 @Id 注解的字段，例如：

```
@Entity
@IdClass(BookId.class)
class Book {
    Book() {}

    @Id
    String isbn;

    @Id
    int printing;

    ...
}
```

但是这种方法存在一个问题：我们应该使用哪个对象来标识一本书，并将其传递给像 `find()` 这样接受标识符的方法呢?

解决方法是编写一个独立的类，其字段与实体的标识符属性匹配。Book 实体的 @IdClass 注解指定要用于该实体的 id 类：

```
class BookId {

    String isbn;
    int printing;

    BookId() {}

    BookId(String isbn, int printing) {
        this.isbn = isbn;
        this.printing = printing;
    }

    @Override
    public boolean equals(Object other) {
        if (other instanceof BookId) {
            BookId bookId = (BookId) other;
            return bookId.isbn.equals(isbn)
                && bookId.printing == printing;
        }
        else {
            return false;
        }
    }

    @Override
    public int hashCode() {
        return isbn.hashCode();
    }
}
```

每个 id 类都应该重写 **equals()** 和 **hashCode()** 方法。

然而，这不是我们首选的方法。相反，我们建议将 **BookId** 类声明为 **@Embeddable** 类型：

```
@Embeddable
class BookId {
```

```
    String isbn;

    int printing;

    BookId() {}

    BookId(String isbn, int printing) {
        this.isbn = isbn;
        this.printing = printing;
    }

    ...
}
```

我们将在下面学到更多关于嵌入对象的知识。

现在，实体类可以使用 **@EmbeddedId** 重用这个定义，不再需要 **@IdClass** 注解：

```
@Entity
class Book {
    Book() {}

    @EmbeddedId
    BookId bookId;

    ...
}
```

这第二种方法消除了一些重复的代码。

无论哪种方法，我们现在都可以使用 `BookId` 来获取 `Book` 的实例：

```
Book book = session.find(Book.class, new BookId(isbn, printing));
```

3.8. 版本属性

一个实体可以有一个由Hibernate用于进行乐观锁检查的属性。版本属性通常是 Integer、Short、Long、LocalDateTime、OffsetDateTime、ZonedDateTime 或 Instant 类型。

```
@Version
LocalDateTime lastUpdated;
```

当一个实体被持久化时，Hibernate会自动为版本属性分配一个值，并在每次实体被更新时自动递增或更新这个值。

如果一个实体没有版本号，通常在映射遗留数据时会发生这种情况，我们仍然可以进行乐观锁。**@OptimisticLocking** 注解允许我们指定通过验证所有字段的值，或者仅验证实体的 DIRTY 字段来检查乐观锁。而 **@OptimisticLock** 注解则允许我们选择性地排除乐观锁定中的某些字段。

我们已经看到的 **@Id** 和 **@Version** 属性只是基本属性的特殊例子。

3.9. 自然标识符属性

即使一个实体有一个代理键，从系统用户的角度来看，总是应该能够写下一组字段，这组字段唯一地标识了实体的一个实例。这组字段被称为它的自然键。在前面，我们考虑了自然键与主键重合的情况。在这里，自然键是实体的第二个唯一键，不同于它的代理主键。

如果你无法确定一个自然键，这可能意味着你需要更仔细地考虑你的数据模型的某些方面。一个实体如果不存在有意义的唯一键，那么就无法确定它在你程序之外的"真实世界"中代表了什么事件或对象。

由于基于自然键检索实体非常常见，Hibernate 提供了一种标记组成实体自然键的属性的方式。每个属性必须使用 `@NaturalId` 注解进行标记。

```
@Entity
class Book {
    Book() {}

    @Id @GeneratedValue
    Long id; // 系统生成的代理键

    @NaturalId
    String isbn; // 属于自然键

    @NaturalId
```

```
    int printing; // 也属于自然键

    ...
}
```

Hibernate 会自动生成关于被这些注解字段映射的列的唯一约束。

考虑使用自然标识符属性来实现 **equals()** 和 **hashCode()** 方法。

进行这些额外的工作的好处是，我们稍后将会看到，我们可以利用优化的自然键查找，这些查找使用了二级缓存。

请注意，即使你已经确定了自然键，我们仍然建议在外键中使用一个生成的代理键，因为这样可以使你的数据模型更容易修改。

译者注：自然主键也比较常见，例如，例如手机电子产品都有一个IMEI码，再比如汽车发动机也有一个唯一vin号码。

3.10. 基本属性

实体的基本属性是映射到关联数据库表的单个列的字段或属性。JPA 规范定义了一组相当有限的基本类型：

表12. JPA 标准基本属性类型

分类	包	类型
原始类型	无	boolean, int, double, 等
基本类型包装类	java.lang	Boolean, Integer, Double, 等
字符串	java.lang	String
任意精度数字类型	java.math	BigInteger, BigDecimal
日期/时间类型	java.time	LocalDate, LocalTime, LocalDateTime, OffsetDateTime, Instant
不推荐使用的日期/时间类型 💀	java.util	Date, Calendar
不推荐使用的 JDBC 日期/时间类型 💀	java.sql	Date, Time, Timestamp
二进制和字符数组	无	byte[], char[]
UUID	java.util	UUID
枚举类型	任何枚举	无
可序列化类型	任何实现 java.io.Serializable 接口的类型	无

> 我们强烈建议使用 `java.time` 包中的类型，而不是任何继承自 `java.util.Date` 的类型。

> 将Java对象序列化并将其二进制表示存储在数据库中通常是错误的。正如我们将很快在嵌入对象中看到的，Hibernate 有处理复杂Java对象的更好方法。

Hibernate 在这个列表中稍微扩展了一下，增加了以下类型：

表13. Hibernate 中的附加基本属性类型

分类	包	类型
附加的日期/时间类型	java.time	Duration, ZoneId, ZoneOffset, Year，甚至 ZonedDateTime
JDBC LOB 类型	java.sql	Blob, Clob, NClob
Java 类对象	java.lang	Class
杂项类型	java.util	Currency, URL, TimeZone

`@Basic` 注解明确指定一个属性是基本属性，但通常不需要，因为属性默认被假定为基本属性。另一方面，如果非原始类型的属性不能为null，强烈建议使用 `@Basic(optional=false)`。

```
@Basic(optional=false)
String firstName;

@Basic(optional=false)
String lastName;

String middleName; // 可能为null
```

请注意，默认情况下，原始类型的属性被假定为 NOT NULL。

> **在JPA中使列非空的方法**
>
> 在JPA中，有两种标准的方法可以为映射的列添加 NOT NULL 约束：
>
> 1. 使用 `@Basic(optional=false)`，或者
> 2. 使用 `@Column(nullable=false)`。
>
> 也许你会想知道它们之间的区别。
>
> 嗯，对于 casual 用户来说，JPA 注解可能并不明显，但实际上它们有两个“层级”：
>
> - `@Entity`、`@Id` 和 `@Basic` 这样的注解属于逻辑层，是当前章节主题的一部分——它们指定了你的Java领域模型的语义，而
> - `@Table` 和 `@Column` 这样的注解属于映射层，是下一章主题的一部分——它们指定了领域模型元素如何映射到关系数据库中的对象。
>
> 信息可以从逻辑层推导到映射层，但不会反向推导。
>
> 现在，`@Column` 注解，我们稍后会详细介绍，属于映射层，因此它的 `nullable` 成员仅影响模式生成（导致在生成的 DDL 中有一个 NOT NULL 约束）。另一方面，`@Basic` 注解属于逻辑层，所以被标记为 `optional=false` 的属性在Hibernate甚至将实体写入数据库之前就会被检查。请注意：
>
> - `optional=false` 意味着 `nullable=false`，但是
> - `nullable=false` 不意味着 `optional=false`。
>
> 因此，我们更倾向于使用 `@Basic(optional=false)` 而不是 `@Column(nullable=false)`。
>
> 但等等！一个更好的解决方案是使用 Bean 验证的 `@NotNull` 注解。只需将 Hibernate Validator 添加到项目构建中，如 “可选依赖性” 中所述。

3.11. 枚举类型

我们在上面的列表中包括了Java枚举。枚举类型被认为是一种基本类型，但由于大多数数据库没有本地的ENUM类型，JPA 提供了特殊的 `@Enumerated` 注解来指定数据库中如何表示枚举值：

- 默认情况下，枚举被存储为一个整数，即它的 **ordinal()** 成员的值。
- 但是，如果属性被标注为 **@Enumerated(STRING)** ，它将被存储为一个字符串，即它的 **name()** 成员的值。

```
// 这里，将枚举编码为整数是有意义的
@Enumerated
@Basic(optional=false)
DayOfWeek dayOfWeek;

// 但通常，将枚举编码为字符串更好
@Enumerated(EnumType.STRING)
@Basic(optional=false)
Status status;
```

在Hibernate 6中，一个被标注为 **@Enumerated(STRING)** 的枚举会被映射为：

- 大多数数据库上的 VARCHAR 列类型，带有 CHECK 约束，或者
- 在MySQL上的 ENUM 列类型。

其他任何枚举都会被映射为一个带有 CHECK 约束的 TINYINT 列。

> JPA 在这里选择了错误的默认值。在大多数情况下，将枚举值的整数编码存储在关系数据中会使其更难以解释。
>
> 即使是 `DayOfWeek`，将其编码为整数也是模糊的。如果你查看 `java.time.DayOfWeek`，你会注意到 `SUNDAY` 被编码为 6。但在我出生的国家，星期天是一周的第一天！
>
> 因此，我们更倾向于对大多数枚举属性使用 **@Enumerated(STRING)** 。

一个有趣的特例是PostgreSQL。Postgres 支持命名的 ENUM 类型，必须使用 DDL CREATE TYPE 语句进行声明。不幸的是，这些 ENUM 类型在语言上不太集成，Postgres JDBC 驱动也不太支持，所以Hibernate默认不使用它们。但是，如果你想在Postgres上使用命名的枚举类型，只需将你的枚举属性标注如下：

```
@JdbcTypeCode(SqlTypes.NAMED_ENUM)
@Basic(optional=false)
Status status;
```

通过提供一个转换器，可以稍微扩展预定义的基本属性类型的有限集。

3.12. 转换器

JPA的`AttributeConverter`负责：

- 将给定的Java类型转换为上面列出的类型之一，和/或
- 在将基本属性值写入数据库或从数据库读取之前，执行任何其他类型的预处理和后处理。

转换器显著扩展了JPA可以处理的属性类型集。

有两种方法可以应用转换器：

- `@Convert`注解将`AttributeConverter`应用于特定的实体属性，或者
- `@Converter`注解（或者，作为替代，`@ConverterRegistration`注解）为特定类型的所有属性注册`AttributeConverter`，以自动应用。

例如，下面的转换器将自动应用于任何类型为`BitSet`的属性，并负责将`BitSet`持久化到类型为`varbinary`的列中：

```
@Converter(autoApply = true)
public static class BitSetConverter implements AttributeConverter<BitSet, byte[]> {
    @Override
    public byte[] convertToDatabaseColumn(BitSet attribute) {
        // convert BitSet to byte[]
        // ...
    }

    @Override
    public BitSet convertToEntityAttribute(byte[] dbData) {
        // convert byte[] to BitSet
        // ...
    }
}
```

另一方面，如果我们不设置`autoApply=true`，那么我们必须使用`@Convert`注解显式应用转换器：

```
@Convert(converter = BitSetConverter.class)
@Basic(optional = false)
BitSet bitset;
```

这一切都很好，但你可能不会感到意外的是，Hibernate超越了JPA所要求的范围。

3.13. 组合基本类型

Hibernate将一个"基本类型"视为两个对象的结合：

- 一个JavaType，它模拟了某个特定Java类的语义，和
- 一个JdbcType，代表JDBC可以理解的SQL类型。

在映射基本属性时，我们可以显式指定JavaType、JdbcType，或者两者都指定。

JavaType

`org.hibernate.type.descriptor.java.JavaType`的实例表示特定的Java类。它能够：

- 比较该类的实例，以确定该类类型的属性是否被修改（脏状态），
- 为该类的实例产生有用的哈希码，
- 将值强制转换为其他类型，特别是在其合作伙伴JdbcType的请求下，将该类的实例转换为其他几种等效的Java表示形式之一。

例如，`IntegerJavaType`知道如何将`Integer`或`int`值转换为`Long`、`BigInteger`和`String`等类型。

我们可以使用`@JavaType`注解显式指定Java类型，但是对于内置的JavaTypes，这是不必要的。

```
@JavaType(LongJavaType.class)  // 不需要，这是long的默认JavaType
long currentTimeMillis;
```

对于用户编写的JavaType，注解更有用：

```
@JavaType(BitSetJavaType.class)
BitSet bitSet;
```

或者，可以使用 `@JavaTypeRegistration` 注解将 `BitSetJavaType` 注册为 `BitSet` 的默认JavaType。

JdbcType

`org.hibernate.type.descriptor.jdbc.JdbcType` 能够从JDBC中读取和写入单个Java类型。

例如，`VarcharJdbcType` 负责：

- 通过调用 `setString()` 将Java字符串写入JDBC PreparedStatements，以及
- 使用 `getString()` 从JDBC ResultSets 读取Java字符串。

通过将 `LongJavaType` 与 `VarcharJdbcType` 配对，我们生成了一个基本类型，将 `Long` 和 `long` 映射到 SQL 类型 `VARCHAR`。

我们可以使用 `@JdbcType` 注解显式指定JDBC类型。

```
@JdbcType(VarcharJdbcType.class)
long currentTimeMillis;
```

或者，我们可以指定JDBC类型代码：

```
@JdbcTypeCode(Types.VARCHAR)
long currentTimeMillis;
```

`@JdbcTypeRegistration` 注解可用于将用户编写的 `JdbcType` 注册为给定SQL类型代码的默认类型。

JDBC类型和JDBC类型代码

JDBC规范定义的类型由类 `java.sql.Types` 中的整数类型代码进行枚举。每个JDBC类型都是SQL中常用类型的抽象。例如，`Types.VARCHAR` 代表 SQL 类型 `VARCHAR`（或者在Oracle上是 `VARCHAR2`）。

由于Hibernate理解的SQL类型比JDBC多，因此在类 `org.hibernate.type.SqlTypes` 中有一个扩展的整数类型代码列表。例如，`SqlTypes.GEOMETRY` 代表空间数据类型 `GEOMETRY`。

AttributeConverter

如果给定的 `JavaType` 不知道如何将其实例转换为其合作伙伴 `JdbcType` 需要的类型，我们必须通过提供JPA `AttributeConverter` 来帮助它执行转换。

例如，要使用 `LongJavaType` 和 `TimestampJdbcType` 形成一个基本类型，我们将提供一个 `AttributeConverter<Long, Timestamp>`。

```
@JdbcType(TimestampJdbcType.class)
@Convert(converter = LongToTimestampConverter.class)
long currentTimeMillis;
```

在这里，让我们停止我们的类比，免得我们开始称这个基本类型为“三人行”。

3.14. 可嵌入对象

可嵌入对象是一个Java类，其状态映射到表的多个列，但它本身没有持久化标识。换句话说，它是一个具有映射属性但没有@Id属性的类。

可嵌入对象只能通过将其分配给实体的属性来使其持久化。由于可嵌入对象没有自己的持久化标识，它的持久性生命周期完全由其所属实体的生命周期决定。

可嵌入类必须用@Embeddable注解而不是@Entity注解。

```
@Embeddable
class Name {

    @Basic(optional=false)
    String firstName;

    @Basic(optional=false)
    String lastName;

    String middleName;

    Name() {}

    Name(String firstName, String middleName, String lastName) {
        this.firstName = firstName;
        this.middleName = middleName;
```

```
        this.lastName = lastName;
    }

    ...
}
```

可嵌入类必须满足实体类满足的相同要求，唯一的例外是可嵌入类没有@Id属性。特别是，它必须有一个没有参数的构造函数。

或者，可嵌入类型可以被定义为Java记录类型：

```
@Embeddable
record Name(String firstName, String middleName, String lastName) {}
```

在这种情况下，对没有参数构造函数的要求放宽了。

不幸的是，截至2023年5月，Java记录类型仍然不能用作@EmbeddedIds。

现在，我们可以将我们的Name类（或记录）用作实体属性的类型：

```
@Entity
class Author {
    @Id @GeneratedValue
    Long id;

    Name name;

    ...
}
```

可嵌入类型可以被嵌套。也就是说，一个@Embeddable类可以有一个属性，其类型本身是另一个@Embeddable类。

JPA提供了@Embedded注解来标识实体的属性，该属性引用可嵌入类型。这个注解是完全可选的，所以我们通常不使用它。

另一方面，对可嵌入类型的引用永远不是多态的。一个@Embeddable类F可能继承第二个@Embeddable类E，但是类型为E的属性将始终引用该具体类E的实例，而不是F的实例。

通常，可嵌入类型以"扁平化"格式存储。它们的属性映射到其父实体的表的列。稍后，在将可嵌入类型映射到UDTs或JSON中，我们将看到一些不同的选项。

可嵌入类型的属性表示Java对象之间的关系，其中一个具有持久化标识，另一个没有持久化标识。我们可以将其视为整体/部分关系。可嵌入对象属于实体，并且不能与其他实体实例共享。它只存在于其父实体存在的时间。

接下来，我们将讨论一种不同类型的关系：Java对象之间具有自己独立持久化标识和持久化生命周期的关系。

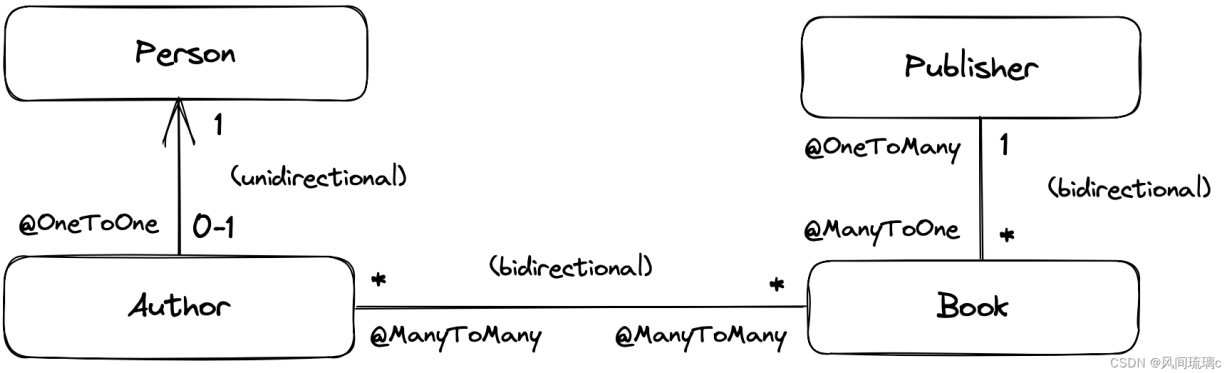
3.15. 关联关系 关联关系是实体之间的关系。我们通常根据它们的多重性进行分类。如果E和F都是实体类，那么：

- 一对一关联将最多一个唯一实例E与最多一个唯一实例F关联。
- 多对一关联将零个或多个E的实例与唯一的F的实例关联。
- 多对多关联将零个或多个E的实例与零个或多个F的实例关联。

实体类之间的关联可以是：

- 单向的，从E到F是可以导航的，但是从F到E不可以。
- 双向的，可以在任何方向上导航。

在上面的数据模型示例中，我们可以看到可能的关联关系：



聪明的观察者可能会注意到，我们作为单向一对一关联呈现的关系可以合理地使用子类型化来表示。这是非常正常的。在完全规范化的关系模型中，一对一关联是我们实现子类型化的常用方式。它与JOINED继承映射策略相关。

有三个用于映射关联的注解：@ManyToOne，@OneToMany和@ManyToMany。它们共享一些常见的注解成员：

成员	解释	默认值
cascade	应该级联到关联实体的持久化操作；CascadeTypes的列表	{}
fetch	关联是否急切加载或者是否可以被代理	@OneToMany和@ManyToMany为LAZY，@ManyToOne为EAGER
targetEntity	关联的实体类	从属性类型声明确定
optional	对于@ManyToOne或@OneToOne关联，关联是否可以为空	true
mappedBy	对于双向关联，映射关联的关联实体的属性	默认情况下，假定关联是单向的

当我们考虑各种关联映射时，我们将解释这些成员的影响。

让我们从最常见的关联多重性开始。

3.16. 多对一

多对一关联是我们可以想象的最基本的关联形式。它在数据库中自然地映射到一个外键。你的域模型中几乎所有的关联都将是这种形式。

稍后，我们将看到如何将多对一关联映射到一个关联表。

@ManyToOne注解标记了关联的"多"方，所以单向的多对一关联看起来像这样：

```
class Book {
    @Id @GeneratedValue
    Long id;

    @ManyToOne(fetch=LAZY)
    Publisher publisher;

    ...
}
```

在这里，Book 表有一个外键列，保存关联的 Publisher 的标识符。

> JPA 中一个非常不幸的特性是，默认情况下 `@ManyToOne` 关联是急加载的。这几乎从不是我们想要的。几乎所有的关联都应该是懒加载的。`fetch=EAGER` 只有在我们确信关联对象有很高的概率在二级缓存中找到时才有意义。除非是这种情况，否则请记得明确指定 `fetch=LAZY`。

大多数时候，我们希望能够方便地在两个方向上导航我们的关联。我们确实需要一种方法来获取给定 Book 的 Publisher，但我们也希望能够获取属于给定出版商的所有 Book。

为了使这个关联双向，我们需要在 Publisher 类中添加一个集合属性，并使用 `@OneToMany` 进行标注。

> Hibernate 在运行时需要代理未获取的关联。因此，多值关联的一侧必须使用接口类型（比如 Set 或 List）声明，而不是使用具体类型（比如 HashSet 或 ArrayList）。

为了清晰地指示这是一个双向关联，并且重用已在 Book 实体中指定的任何映射信息，我们必须使用 `mappedBy` 注解成员来指向 Book.publisher。

```
@Entity
class Publisher {
    @Id @GeneratedValue
    Long id;

    @OneToMany(mappedBy="publisher")
    Set<Book> books;

    ...
}
```

Publisher.books 字段被称为关联的非拥有方。

现在，我们非常讨厌 mappedBy 引用关联的拥有方的字符串类型。幸运的是，Metamodel 生成器给了我们一种使其类型安全一些的方式：

```
@OneToMany(mappedBy=Book_.PUBLISHER)  // 习惯用这种方式!
Set<Book> books;
```

我们将在介绍的其余部分中使用这种方法。

要修改双向关联，我们必须改变拥有方。

> 对关联的非拥有方所做的更改永远不会同步到数据库。如果我们想要在数据库中修改关联，必须从拥有方进行修改。在这里，我们必须设置 Book.publisher。
>
> 实际上，通常需要同时改变双向关联的两边。例如，如果集合 Publisher.books 存储在二级缓存中，我们还必须修改集合，以确保二级缓存与数据库保持同步。

也就是说，更新非拥有方并不是硬性要求，至少如果你确信自己知道在做什么的话。

> 原则上，Hibernate 允许你有一个单向一对多关联，也就是，在另一侧没有匹配的 `@ManyToOne`。实际上，这种映射是不自然的，而且效果并不好。应该避免使用它。

在这里，我们使用了 Set 作为集合的类型，但是 Hibernate 也允许在这里使用 List 或 Collection，语义上几乎没有区别。特别是，List 不能包含重复元素，并且其顺序不会被持久化。

```
@OneToMany(mappedBy=Book_.PUBLISHER)
Collection<Book> books;
```

> Set、List 还是 Collection?
>
> 映射到外键的一对多关联永远不能包含重复元素，因此 Set 看起来是最语义正确的 Java 集合类型，因此在 Hibernate 社区中，这是常规做法。
>
> 使用 Set 的关键是，我们必须仔细确保 Book 有一个高质量的 `equals()` 和 `hashCode()` 实现。现在，这并不一定是一件坏事，因为一个高质量的 `equals()` 独立来说也是有用的。
>
> 但如果我们使用 Collection 或者 List 呢？那么我们的代码就不太依赖于 `equals()` 和 `hashCode()` 的实现方式了。
>
> ```
> @Entityclass Person {    @Id @GeneratedValue    Long id;     @OneToOne(mappedBy = Author_.PERSON)
>  Author author;     ...}
> ```

在过去，我们可能过于教条地推荐使用 Set。现在呢？我想我们很高兴让你们自己决定。事后看来，我们本可以做得更多，以明确这一直都是一个可行的选择。

3.17. 一对一关联（第一种方式）

最简单的一对一关联几乎与 `@ManyToOne` 关联完全相同，唯一的区别是它映射到一个带有 UNIQUE 约束的外键列。

> 稍后，我们将看到如何将一对一关联映射到一个关联表。

一对一关联必须使用 `@OneToOne` 进行注解：

```
@Entity
class Author {
    @Id @GeneratedValue
    Long id;

    @OneToOne(optional=false, fetch=LAZY)
    Person author;

    ...
}
```

在这里，Author 表有一个外键列，保存关联的 Person 的标识符。

> 一对一关联通常建模为“类型”的关系。在我们的例子中，Author 是 Person 的一种类型。在 Java 中表示“类型”的关系的另一种替代方式，通常更自然，是通过实体类继承。

我们可以通过在 Person 实体中添加对 Author 的引用使这个关联双向：

```
@Entity
class Person {
    @Id @GeneratedValue
    Long id;

    @OneToOne(mappedBy = Author_.PERSON)
    Author author;

    ...
}
```

Person.author 是非拥有方，因为它被标记为 `mappedBy`。

> 一对一关联的懒加载

> 请注意，我们没有声明非拥有方的关联使用 `fetch=LAZY`。这是因为：
>
> 1. 不是每个 Person 都有一个关联的 Author，和
> 2. 外键存储在 Author 映射的表中，而不是 Person 映射的表中。
>
> 因此，Hibernate 无法在不获取关联的 Author 的情况下判断从 Person 到 Author 的引用是否为 null。
>
> 另一方面，如果每个 Person 都是一个 Author，也就是说，如果这个关联是非可选的，我们就不需要考虑 null 引用的可能性，我们会这样映射它：
>
> ```
> @OneToOne(optional=false, mappedBy = Author_.PERSON, fetch=LAZY)
> Author author;
> ```

这并不是唯一一种一对一关联的方式。

3.18. 一对一关联（第二种方式）

表示这种关系的一种更加优雅的方式是在两个表之间共享一个主键。

要使用这种方法，Author 类必须这样进行注解：

```
@Entity
class Author {
    @Id
    Long id;

    @OneToOne(optional=false, fetch=LAZY)
    @MapsId
    Person author;

    ...
}
```

请注意，与之前的映射相比：

- `@Id` 属性不再是 `@GeneratedValue`，而是
- 相反，author 关联被注解为 `@MapsId`。

这让 Hibernate 知道与 Person 的关联是 Author 主键值的来源。

在这里，Author 表中没有额外的外键列，因为 id 列保存了 Person 的标识符。也就是说，Author 表的主键不仅充当了自身的主键，还作为外键引用到了 Person 表。

Person 类保持不变。如果关联是双向的，我们像之前一样，对非拥有方进行注解 `@OneToOne(mappedBy = Author_.PERSON)`。

3.19. 多对多关联

单向多对多关联被表示为一个集合属性。它总是映射到数据库中的一个单独的关联表。

通常情况下，一个多对多关联最终会变成一个伪装成实体的关联。

> 假设我们从 Author 和 Book 之间一个很干净的多对多关联开始。后来，很可能我们会发现一些额外的信息与关联相关，因此关联表需要一些额外的列。
>
> 例如，假设我们需要报告每个作者对一本书的贡献百分比。这种信息自然属于关联表。我们不能将其轻松地存储为 Book 的属性，也不能将其存储为 Author 的属性。
>
> 当发生这种情况时，我们需要改变我们的 Java 模型，通常引入一个新的实体类，直接映射到关联表。在我们的例子中，我们可以将这个实体命名为 BookAuthorship，它会有 @OneToMany 关联到 Author 和 Book，还有贡献属性。
>
> 我们可以通过简单地从一开始就避免使用 `@ManyToMany` 来规避这种“发现”引起的混乱。使用一个中间实体来表示每一个（或者至少几乎每一个）逻辑上的多对多关联几乎没有什么坏处。

多对多关联必须使用 `@ManyToMany` 进行注解：

```
@Entity
class Book {
    @Id @GeneratedValue
    Long id;

    @ManyToMany
    Set<Author> authors;

    ...
}
```

如果关联是双向的，我们在 Book 中添加一个看起来非常相似的属性，但是这次我们必须指定 `mappedBy`，以表示这是关联的非拥有方：

```
@Entity
class Book {
    @Id @GeneratedValue
    Long id;

    @ManyToMany(mappedBy=Author_.BOOKS)
    Set<Author> authors;

    ...
}
```

记住，如果我们想修改集合，我们必须改变拥有方。

我们再次使用 Set 来表示关联。与之前一样，我们有使用 Collection 或 List 的选项。但是在这种情况下，这确实会影响到关联的语义。

> 一个以 Collection 或 List 表示的多对多关联可能包含重复元素。然而，与之前一样，元素的顺序不是持久化的。也就是说，这个集合是一个包，而不是一个集合。

3.20. 基本值和可嵌入对象的集合

到目前为止，我们已经看到了以下种类的实体属性：

实体属性种类	引用类型	多重性	示例
基本类型的单值属性	非实体	最多一个	`@Basic String name`
可嵌入类型的单值属性	非实体	最多一个	`@Embedded Name name`
单值关联	实体	最多一个	`@ManyToOne Publisher publisher` `@OneToOne Person person`
多值关联	实体	零个或多个	`@OneToMany Set<Book> books` `@ManyToMany Set<Author> authors`

在这个分类中，你可能会问：Hibernate 是否有基本类型或可嵌入类型的多值属性呢？

实际上，在两种特殊情况下，Hibernate 已经提供了支持。首先，我们要记住，JPA 将 byte[] 和 char[] 数组视为基本类型。Hibernate 会将 byte[] 或 char[] 数组持久化到 VARBINARY 或 VARCHAR 列中。

但在这一节中，我们真正关心的是除了这两种特殊情况之外的情况。那么，除了 byte[] 和 char[] 之外，Hibernate 是否支持基本类型或可嵌入类型的多值属性呢？

答案是肯定的。实际上，有两种不同的方式来处理这样的集合，可以将其映射为：

1. SQL ARRAY 类型的列（假设数据库支持 ARRAY 类型）；或者
2. 单独的表。

因此，我们可以将我们的分类扩展为：

实体属性种类	引用类型	多重性	示例
byte[] 和 char[] 数组	非实体	零个或多个	`byte[] image` `char[] text`
基本类型元素的集合	非实体	零个或多个	`@Array String[] names` `@ElementCollection Set<String> names`
可嵌入类型元素的集合	非实体	零个或多个	`@ElementCollection Set<Name> names`

实际上，这里有两种新的映射方式：`@Array` 映射和 `@ElementCollection` 映射。

> 这些映射方式被过度使用。
>
> 在我们的实体类中，有时我们认为使用基本类型的值集合是合适的。但这种情况很少见。几乎每一个多值关联应该映射到独立表格之间的外键关联。而且几乎每一个表格都应该由一个实体类来映射。
>
> 在接下来的两个小节中，我们将介绍的特性，初学者使用得比专家多得多。所以如果你是初学者，最好在现阶段避免使用这些特性，可以节省很多麻烦。

我们将首先讨论 `@Array` 映射。

3.21. 映射到 SQL 数组的集合

让我们考虑一个在一周的某些天重复的日历事件。我们可以在我们的 Event 实体中将它表示为一个类型为 DayOfWeek[] 或 List 的属性。由于这个数组或列表的元素数量上限是 7，这是使用 ARRAY 类型列的一个合理案例。很难想象将这个集合存储在一个单独的表中有多大的价值。

> **学会不讨厌 SQL 数组**
>
> 很长一段时间，我们认为数组是一种奇怪和不完善的关系模型，但最近我们意识到这种观点过于保守。实际上，我们可以选择将 SQL ARRAY 类型视为 VARCHAR 和 VARBINARY 对通用"元素"类型的一种泛化。从这个角度看，SQL 数组看起来相当有吸引力，至少对于某些问题来说是这样的。如果我们可以将 byte[] 映射到 VARBINARY(255)，那么我们为什么要回避将 DayOfWeek[] 映射到 TINYINT ARRAY[7]?

不幸的是，JPA 没有定义一种标准的方法来映射 SQL 数组，但在 Hibernate 中，我们可以这样做：

```
@Entity
class Event {
    @Id @GeneratedValue
    Long id;
    ...
    @Array(length=7)
    DayOfWeek[] daysOfWeek;  // 存储为 SQL ARRAY 类型
    ...
}
```

`@Array` 注解是可选的，但是限制数据库分配给 ARRAY 列的存储空间是非常重要的。

> 现在要注意的是，并不是每个数据库都有 SQL ARRAY 类型，而且一些具有 ARRAY 类型的数据库也不允许它被用作列类型。
>
> 特别是，DB2 和 SQL Server 都没有 ARRAY 类型的列。在这些数据库上，Hibernate 采用了一个更糟糕的方案：它使用 Java 序列化将数组编码为二进制表示，并将二进制流存储在 VARBINARY 列中。很显然，这是糟糕的。您可以通过在属性上添加 `@JdbcTypeCode(SqlTypes.JSON)` 注解，将数组序列化为 JSON 格式而不是二进制格式，来让 Hibernate 做得稍微好一点。但是在这一点上，最好承认失败，改用 `@ElementCollection`。

另外，我们还可以将这个数组或列表存储在一个单独的表中。

3.22. 映射到单独表的集合

JPA 确实定义了一种将集合映射到辅助表的标准方法：`@ElementCollection` 注解。

```
@Entity
class Event {
    @Id @GeneratedValue
    Long id;
    ...
    @ElementCollection
    DayOfWeek[] daysOfWeek;  // 存储在一个专用表中
    ...
}
```

实际上，在这里我们不应该使用数组，因为数组类型不能被代理，因此 JPA 规范甚至不支持它们。相反，我们应该使用 Set、List 或 Map。

```
@Entity
class Event {
    @Id @GeneratedValue
    Long id;
    ...
    @ElementCollection
    List<DayOfWeek> daysOfWeek;  // 存储在一个专用表中
    ...
}
```

在这里，每个集合元素都被存储为辅助表的一个单独行。默认情况下，该表的定义如下：

```
create table Event_daysOfWeek (
    Event_id bigint not null,
    daysOfWeek tinyint check (daysOfWeek between 0 and 6),
    daysOfWeek_ORDER integer not null,
    primary key (Event_id, daysOfWeek_ORDER)
)
```

这是可以接受的，但这仍然是一种我们更愿意避免的映射方式。

`@ElementCollection` 是我们最不喜欢的 JPA 特性之一。甚至注解的名称都不好。

上述代码会生成一个带有三个列的表格：

1. Event 表的外键，
2. 一个编码枚举的 TINYINT，以及
3. 一个编码数组中元素顺序的 INTEGER。

它不是使用一个代理主键，而是使用一个由 Event 的外键和顺序列组成的复合主键。

当——不可避免地——我们发现需要在表中添加第四列时，我们的 Java 代码必须完全改变。很可能，我们会意识到我们最终需要添加一个单独的实体。因此，这种映射在面对数据模型的微小变化时并不够健壮。

关于 "元素集合"，我们还可以说很多，但我们不再继续，因为我们不希望你搬起石头砸自己的脚。

3.23. 注解总结

让我们暂停一下，回顾一下我们到目前为止遇到的注解。

表 15. 声明实体和可嵌入类型

Annotation	Purpose	JPA-standard
@Entity	声明一个实体类	✓
@MappedSuperclass	声明一个非实体类，其映射属性被实体类继承	✓
@Embeddable	声明一个可嵌入类型	✓
@IdClass	为具有多个 @Id 属性的实体声明标识符类	✓

表 16. 声明基本和嵌入属性

Annotation	Purpose	JPA-standard
@Id	声明基本类型的标识符属性	✓
@Version	声明版本属性	✓
@Basic	声明基本属性	默认即为 @Basic
@EmbeddedId	声明嵌入类型的标识符属性	✓
@Embedded	声明嵌入类型的属性	推断即可
@Enumerated	声明枚举类型的属性并指定其编码方式	推断即可
@Array	声明一个属性映射到 SQL ARRAY，并指定长度	推断即可
@ElementCollection	声明一个集合映射到专用表	✓

表 17. 转换器和组合基本类型

Annotation	Purpose	JPA-standard
@Converter	注册一个 AttributeConverter	✓
@Convert	将转换器应用于属性	✓
@JavaType	明确指定基本属性的 JavaType 实现	✕
@JdbcType	明确指定基本属性的 JdbcType 实现	✕
@JdbcTypeCode	明确指定用于确定基本属性的 JdbcType 的 JDBC 类型代码	✕
@JavaTypeRegistration	为给定的 Java 类型注册一个 JavaType	✕
@JdbcTypeRegistration	为给定的 JDBC 类型代码注册一个 JdbcType	✕

表 18. 系统生成的标识符

Annotation	Purpose	JPA-standard
@GeneratedValue	指定标识符由系统生成	✓
@SequenceGenerator	定义一个由数据库序列支持的 ID 生成器	✓
@TableGenerator	定义一个由数据库表支持的 ID 生成器	✓
@IdGeneratorType	为其注释的每个 @Id 属性关联一个自定义生成器的注解	✕
@ValueGenerationType	为其注释的每个 @Basic 属性关联一个自定义生成器的注解	✕

表 19. 声明实体关联

Annotation	Purpose	JPA-standard
@ManyToOne	声明多对一关联的单值方（拥有方）	✓
@OneToMany	声明多对一关联的多值方（非拥有方）	✓
@ManyToMany	声明一对一关联的任意一方	✓
@OneToOne	声明一对一关联的任意一方	✓
@MapsId	声明 @OneToOne 关联的拥有方映射到主键列	✓

哇！这已经是很多的注解了，而且我们甚至还没有开始使用用于对象关系映射的注解！

3.24. equals() 和 hashCode()

实体类应该重写 equals() 和 hashCode() 方法，特别是当关联关系被表示为集合时。

对于刚接触 Hibernate 或 JPA 的人来说，关于 hashCode() 应该包含哪些字段常常感到困惑。而那些更有经验的人则会对哪种方法是唯一正确的方式进行宗教般的争论。事实上，并没有唯一正确的方法，但有一些约束。因此，请牢记以下原则：

1. 不应该在 hashCode() 方法中包含可变字段，因为这会导致每次字段变化时都需要重新计算包含该实体的所有集合的哈希值。
2. 在 hashCode() 方法中包含生成的标识符（代理键）并不完全是错误的，但是由于标识符在实体实例持久化之前是不存在的，所以你必须非常小心，在标识符生成之前，不要将实体放入任何基于哈希的集合中。因此，我们建议不要将任何数据库生成的字段包含在 hashCode() 方法中。
3. 可以在 hashCode() 方法中包含任何不可变的、非生成的字段。

因此，我们建议为每个实体类确定一个自然键，即一组字段，从程序的数据模型的角度来看，它们能唯一标识实体的一个实例。这个自然键应该对应数据库上的唯一约束，并且应该包含在 equals() 和 hashCode() 方法中。

在下面的例子中，equals() 和 hashCode() 方法基于 @NaturalId 注解的字段：

```
@Entity
class Book {

    @Id @GeneratedValue
    Long id;

    @NaturalId
    @Basic(optional=false)
    String isbn;

    ...

    @Override
    public boolean equals(Object other) {
        return other instanceof Book
            && ((Book) other).isbn.equals(isbn);
    }
    @Override
    public int hashCode() {
        return isbn.hashCode();
    }
}
```

尽管如此，基于实体的生成标识符实现的 equals() 和 hashCode() 方法也是可行的，只要你小心处理。

Hibernate 6 中文文档（四）

目录

4.对象/关系映射

4.1. 映射实体继承层次

4.2. 映射到数据库表格

4.3. 实体映射到表格

4.4. 关联映射到表格

4.5. 映射到列

4.6. 将基本属性映射到列

4.7. 将关联映射到外键列

4.8. 映射表格之主键连接

4.9. 列长度和自适应列类型

4.10. 大对象（LOBs）

4.11. 映射可嵌入类型到 UDTs 或 JSON

4.12. SQL列类型映射总结

4.14. 派生标识

4.15. 添加约束

4.对象/关系映射

在给定了一个领域模型（即一组带有我们在前一章中介绍的所有注解的实体类）之后，Hibernate 将愉快地根据它推断出完整的关系型模式，甚至在你礼貌地询问时，还可以将它导出到你的数据库中。

生成的模式将会非常合理和可行，尽管如果你仔细观察，你可能会发现一些问题。例如，每个 VARCHAR 列的长度都是相同的，VARCHAR(255)。

但是，我刚刚描述的这个过程，我们称之为自上而下的映射，实际上并不适用于最常见的对象/关系映射的使用场景。通常，Java 类通常不会先于关系模式存在。通常情况下，我们已经有了一个关系型模式，并且我们正在围绕这个模式构建我们的领域模型。这被称为自下而上的映射。

开发人员通常将现有的关系数据库称为"遗留"数据。这常常让人联想到那些糟糕的"遗留应用程序"，可能是用 COBOL 或其他什么语言写的。但是，遗留数据具有很大的价值，学会如何处理它们非常重要。

特别是在自下而上的映射过程中，我们经常需要自定义推断出的对象/关系映射。这是一个有点乏味的主题，所以我们不打算在这方面花太多篇幅。相反，我们将快速浏览一下最重要的映射注解。

Hibernate 的 SQL 命名约定 计算机上的小写字母已经有很长一段时间了。大多数开发人员早就学会了在 MixedCase、camelCase，甚至 snake_case 中编写的文本比在 SHOUTYCASE 中编写的文本更容易阅读。这一点不仅适用于 SQL，也适用于任何其他语言。

因此，20 多年来，Hibernate 项目上的约定是：

- 查询语言标识符使用小写编写，
- 表名使用 MixedCase 编写，
- 列名使用 camelCase 编写。

也就是说，我们简单地采用了 Java 的优秀约定，并将其应用到了 SQL 中。

当然，我们无法强制你遵循这个约定，即使我们愿意。实际上，你可以很容易地编写一个 `PhysicalNamingStrategy`，如果你愿意的话，可以将表名和列名设置得像这样 ALL UGLY AND SHOUTY。但是，默认情况下，这是 Hibernate 遵循的约定，而且实际上是一个相当合理的约定。

4.1. 映射实体继承层次

在实体类继承中，我们看到实体类可以存在于继承层次结构中。有三种基本策略可以将实体层次结构映射到关系数据库表中。我们将它们放入一个表格中，以便更容易比较它们之间的差异点。

表格 20. 实体继承映射策略

策略	映射	多态查询	约束	规范化	何时使用
`SINGLE_TABLE`	将层次结构中的每个类映射到同一张表，并使用鉴别器列的值确定每行代表哪个具体类。	只需查询一张表即可检索到给定类的实例。	由子类声明的属性映射到没有 NOT NULL 约束的列。	任何关联都可以有外键约束。	子类数据是非规范化的。
`JOINED`	将层次结构中的每个类映射到一个单独的表，但每个表只映射该类自身声明的属性。	必须通过以下方式联接该类的表：与其超类映射的所有表以及其子类映射的所有表。	任何属性都可以映射到带有 NOT NULL 约束的列。	任何关联都可以有外键约束。	表格是规范化的。
`TABLE_PER_CLASS`	将层次结构中的每个具体类映射到一个单独的表，但将所有继承属性都非规范化到表中。	必须在该类的表和其子类映射的所有表之间执行 UNION 操作。	针对超类的关联在数据库中不能有相应的外键约束。	任何属性都可以映射到带有 NOT NULL 约束的列。	超类数据是非规范化的。

这三种映射策略由 `InheritanceType` 枚举定义。我们使用 `@Inheritance` 注解来指定继承映射策略。

对于带有鉴别器列的映射，我们应该：

- 通过在根实体上使用 `@DiscriminatorColumn` 注解来指定鉴别器列的名称和类型，并且
- 通过在层次结构中的每个实体上使用 `@DiscriminatorValue` 注解来指定鉴别器的值。

对于单表继承，我们总是需要一个鉴别器：

```
@Entity
@DiscriminatorColumn(discriminatorType=CHAR, name="kind")
@DiscriminatorValue('P')
class Person { ... }

@Entity
@DiscriminatorValue('A')
class Author { ... }
```

我们不需要明确指定 `@Inheritance(strategy=SINGLE_TABLE)`，因为那是默认值。

对于 `JOINED` 继承，我们不需要鉴别器：

```
@Entity
@Inheritance(strategy=JOINED)
class Person { ... }

@Entity
class Author { ... }
```

然而，如果我们愿意的话，可以添加一个鉴别器列，在这种情况下，多态查询的生成 SQL 将会稍微简单一些。

类似地，对于 `TABLE_PER_CLASS` 继承，我们有：

```
@Entity
@Inheritance(strategy=TABLE_PER_CLASS)
class Person { ... }

@Entity
class Author { ... }
```

Hibernate 不允许 `TABLE_PER_CLASS` 继承映射使用鉴别器列，因为它们没有意义，也没有优势。

请注意，在这种情况下，像这样的多态关联：

```
@ManyToOne
Person person;
```

是一个不好的主意，因为无法创建同时针对两个映射表的外键约束。

4.2. 映射到数据库表格

下面的注解精确指定了领域模型元素如何映射到关系模型的表格：

表格 21. 映射表格的注解

注解	用途
`@Table`	将实体类映射到其主表格。
`@SecondaryTable`	为实体类定义一个附加表格。
`@JoinTable`	将多对多或多对一关联映射到其关联表格。
`@CollectionTable`	将 `@ElementCollection` 映射到其表格。

前两个注解用于将实体映射到其主表格，可选择性地，还可以映射到一个或多个附加表格。

4.3. 实体映射到表格

默认情况下，一个实体映射到一个单独的表格，可以使用 `@Table` 注解指定表格：

```
@Entity
@Table(name="People")
class Person { ... }
```

然而，`@SecondaryTable` 注解允许我们将其属性分散到多个附加表格中：

```
@Entity@Table(name="Books")
@SecondaryTable(name="Editions")
class Book { ... }
```

`@Table` 注解不仅可以指定一个名称：

表格 22. @Table 注解成员

注解成员	用途
`name`	映射表格的名称
`schema` 💀	表格所属的模式（schema）
`catalog` 💀	表格所属的目录（catalog）
`uniqueConstraints`	一个或多个 `@UniqueConstraint` 注解，声明多列的唯一约束
`indexes`	一个或多个 `@Index` 注解，每个声明一个索引

只有当领域模型分布在多个模式中时，通过注解显式指定模式才有意义。

否则，在 `@Table` 注解中硬编码模式（或目录）是个坏主意。相反地：

- 可以设置配置属性 `hibernate.default_schema`（或 `hibernate.default_catalog`），或者
- 在 JDBC 连接 URL 中简单地指定模式。

`@SecondaryTable` 注解更有趣：

表格 23. @SecondaryTable 注解成员

注解成员	用途
`name`	映射表格的名称
`schema` 💀	表格所属的模式（schema）
`catalog` 💀	表格所属的目录（catalog）
`uniqueConstraints`	一个或多个 `@UniqueConstraint` 注解，声明多列的唯一约束
`indexes`	一个或多个 `@Index` 注解，每个声明一个索引
`pkJoinColumns`	一个或多个 `@PrimaryKeyJoinColumn` 注解，指定主键列映射
`foreignKey`	一个 `@ForeignKey` 注解，指定 `@PrimaryKeyJoinColumns` 的外键约束

在 `SINGLE_TABLE` 实体继承层次结构中，在子类上使用 `@SecondaryTable` 注解可以得到一种 SINGLE_TABLE 和 JOINED 继承的混合形式。

4.4. 关联映射到表格

`@JoinTable` 注解指定了一个关联表格，即一个保存两个关联实体的外键的表格。通常，这个注解与 `@ManyToMany` 关联一起使用：

```
@Entity
class Book {
    ...

    @ManyToMany
    @JoinTable(name="BooksAuthors")
    Set<Author> authors;

    ...
}
```

但是，它甚至也可以用于将 `@ManyToOne` 或 `@OneToOne` 关联映射到关联表格。

```
@Entity
class Book {
    ...

    @ManyToOne(fetch=LAZY)
    @JoinTable(name="BookPublisher")
    Publisher publisher;

    ...
}
```

在这里，关联表格的一个列上应该有一个唯一约束。

```
@Entity
class Author {
    ...

    @OneToOne(optional=false, fetch=LAZY)
    @JoinTable(name="AuthorPerson")
    Person author;

    ...
}
```

在这里，关联表格的两个列上应该有唯一约束。

表格 24. @JoinTable 注解成员

注解成员	用途
`name`	映射的关联表格的名称
`schema` 💀	表格所属的模式（schema）
`catalog` 💀	表格所属的目录（catalog）
`uniqueConstraints`	一个或多个 `@UniqueConstraint` 注解，声明多列的唯一约束
`indexes`	一个或多个 `@Index` 注解，每个声明一个索引
`joinColumns`	一个或多个 `@JoinColumn` 注解，指定关联方表格中的外键列映射
`inverseJoinColumns`	一个或多个 `@JoinColumn` 注解，指定非关联方表格中的外键列映射
`foreignKey`	一个 `@ForeignKey` 注解，指定 `joinColumns` 上的外键约束名称
`inverseForeignKey`	一个 `@ForeignKey` 注解，指定 `inverseJoinColumns` 上的外键约束名称

为了更好地理解这些注解，我们首先必须讨论一般的列映射。

4.5. 映射到列

以下注解指定了领域模型元素如何映射到关系模型表格的列：

表格 25. 映射列的注解

注解	用途
`@Column`	将属性映射到一个列。
`@JoinColumn`	将关联映射到一个外键列。
`@PrimaryKeyJoinColumn`	将用于将次表格与其主表格、或在 JOINED 继承中将子类表格与其根类表格连接的主键进行映射。
`@OrderColumn`	指定用于维护列表顺序的列。
`@MapKeyColumn`	指定用于持久化映射键的列。

我们使用 `@Column` 注解来映射基本属性。

4.6. 将基本属性映射到列

`@Column` 注解不仅仅用于指定列名。

表格 26. @Column 注解成员

注解成员	用途
`name`	映射的列的名称
`table`	此列所属的表格的名称
`length`	VARCHAR、CHAR 或 VARBINARY 列类型的长度
`precision`	FLOAT、DECIMAL、NUMERIC、TIME 或 TIMESTAMP 列类型的小数位数
`scale`	DECIMAL 或 NUMERIC 列类型的小数位数，即小数点右边的精度
`unique`	列是否具有 UNIQUE 约束
`nullable`	列是否具有 NOT NULL 约束
`insertable`	列是否应该出现在生成的 SQL INSERT 语句中
`updatable`	列是否应该出现在生成的 SQL UPDATE 语句中
`columnDefinition` 💀	应该用于声明列的 DDL 片段

我们不再建议使用 `columnDefinition`，因为它会导致不可移植的 DDL。Hibernate 有更好的方法来定制生成的 DDL，使用这些方法可以在不同的数据库中实现可移植性。

在这里，我们看到了使用 `@Column` 注解的四种不同方式：

```
@Entity
@Table(name="Books")
@SecondaryTable(name="Editions")
class Book {
    @Id @GeneratedValue
    @Column(name="bookId") // 自定义列名
    Long id;

    @Column(length=100, nullable=false) // 将列声明为 VARCHAR(100) NOT NULL
    String title;

    @Column(length=17, unique=true, nullable=false) // 将列声明为 VARCHAR(17) NOT NULL UNIQUE
    String isbn;

    @Column(table="Editions", updatable=false) // 列属于次表格，并且不会被更新
    int edition;
}
```

我们不使用 \`@Column\` 来映射关联。

4.7. 将关联映射到外键列

`@JoinColumn` 注解用于自定义外键列。

表格 27. @JoinColumn 注解成员

注解成员	用途
`name`	映射的外键列的名称
`table`	此列所属的表格的名称
`referencedColumnName`	映射的外键列引用的列的名称
`unique`	列是否具有 UNIQUE 约束
`nullable`	列是否具有 NOT NULL 约束
`insertable`	列是否应该出现在生成的 SQL INSERT 语句中
`updatable`	列是否应该出现在生成的 SQL UPDATE 语句中
`columnDefinition` 💀	应该用于声明列的 DDL 片段
`foreignKey`	一个 `@ForeignKey` 注解，指定 FOREIGN KEY 约束的名称

一个外键列不一定要引用被关联表格的主键。它可以引用被关联实体的任何其他唯一键，甚至是次表格的唯一键。

在这里，我们看到了如何使用 `@JoinColumn` 定义一个 `@ManyToOne` 关联，将外键列映射到 Book 的 `@NaturalId`：

```
@Entity
@Table(name="Items")
class Item {
    ...

    @ManyToOne(optional=false)
    @JoinColumn(name = "bookIsbn", referencedColumnName = "isbn",
                foreignKey = @ForeignKey(name="ItemsToBooksBySsn"))
    Book book;

    ...
}
```

如果这让你感到困惑：

- `bookIsbn` 是 `Items` 表格中外键列的名称，
- 它引用了 `Books` 表格的唯一键 `isbn`，并且
- 它有一个名为 `ItemsToBooksBySsn` 的外键约束。

请注意，`foreignKey` 成员是完全可选的，只影响 DDL 生成。

如果不使用 `@ForeignKey` 显式指定名称，Hibernate 将生成一个相当丑陋的名称。这是因为某些数据库上外键名称的最大长度受到极大限制，我们需要避免冲突。公平地说，如果你只是用生成的 DDL 进行测试，那么这是完全可以接受的。

对于复合外键，我们可能有多个 `@JoinColumn` 注解：

```
@Entity
@Table(name="Items")
class Item {
    ...

    @ManyToOne(optional=false)
    @JoinColumn(name = "bookIsbn", referencedColumnName = "isbn")
    @JoinColumn(name = "bookPrinting", referencedColumnName = "printing")
    Book book;

    ...
}
```

如果我们需要指定 `@ForeignKey`，这会变得有点混乱：

```
@Entity
@Table(name="Items")
class Item {
    ...

    @ManyToOne(optional=false)
    @JoinColumns(value = {@JoinColumn(name = "bookIsbn", referencedColumnName = "isbn"),
                          @JoinColumn(name = "bookPrinting", referencedColumnName = "printing")},
                 foreignKey = @ForeignKey(name="ItemsToBooksBySsn"))
    Book book;

    ...
}
```

对于映射到 `@JoinTable` 的关联，获取关联需要两个连接，所以我们必须在 `@JoinTable` 注解内声明 `@JoinColumns`：

```
@Entity
class Book {
    @Id @GeneratedValue
    Long id;

    @ManyToMany
    @JoinTable(joinColumns=@JoinColumn(name="bookId"),
               inverseJoinColumns=@joinColumn(name="authorId"),
               foreignKey=@ForeignKey(name="BooksToAuthors"))
    Set<Author> authors;

    ...
}
```

同样，`foreignKey` 成员是可选的。

4.8. 映射表格之主键连接

`@PrimaryKeyJoinColumn` 是一种专用注解，用于映射：

- `@SecondaryTable` 的主键列，它同时也是一个外键，引用主表格；
- 在 `JOINED` 继承层次结构中，映射子类表格的主键列，它同时也是一个外键，引用由根实体映射的主表格。

表格 28. @PrimaryKeyJoinColumn 注解成员

注解成员	用途
`name`	映射的外键列的名称
`referencedColumnName`	映射的外键列引用的列的名称
`columnDefinition` 💀	应该用于声明列的 DDL 片段
`foreignKey`	一个 `@ForeignKey` 注解，指定 FOREIGN KEY 约束的名称

在映射子类表格的主键时，我们将 `@PrimaryKeyJoinColumn` 注解放在实体类上：

```
@Entity
@Table(name="People")
@Inheritance(strategy=JOINED)
class Person { ... }

@Entity
@Table(name="Authors")
@PrimaryKeyJoinColumn(name="personId") // Authors 表格的主键
class Author { ... }
```

但是，在映射次表格的主键时，`@PrimaryKeyJoinColumn` 注解必须位于 `@SecondaryTable` 注解内部：

```
@Entity
@Table(name="Books")
@SecondaryTable(name="Editions",
                pkJoinColumns = @PrimaryKeyJoinColumn(name="bookId")) // Editions 表格的主键
class Book {
    @Id @GeneratedValue
    @Column(name="bookId") // Books 表格的主键名称
    Long id;

    ...
}
```

4.9. 列长度和自适应列类型

Hibernate 根据 `@Column` 注解指定的列长度自动调整生成的 DDL 中的列类型。所以，通常情况下，我们不需要显式指定一个列应该是 TEXT 或 CLOB 类型，也不用担心在 MySQL 上会出现 TINYTEXT、MEDIUMTEXT、TEXT、LONGTEXT 这样的类型，因为Hibernate会根据需要选择其中一个类型，以适应我们指定的字符串长度。

在这里，`Length` 类中定义的常量值非常有用：

表格 29. 预定义的列长度

常量	值	描述
`DEFAULT`	`255`	当没有显式指定长度时，VARCHAR 或 VARBINARY 列的默认长度
`LONG`	`32600`	VARCHAR 或 VARBINARY 列上允许的最大列长度，在 Hibernate 支持的每个数据库上都允许
`LONG16`	`32767`	使用 16 位表示的最大长度（但是对于某些数据库的 VARCHAR 或 VARBINARY 列来说，这个长度太大了）
`LONG32`	`2147483647`	Java 字符串的最大长度

我们可以在 `@Column` 注解中使用这些常量：

```
@Column(length=LONG)
String text;

@Column(length=LONG32)
byte[] binaryData;
```

通常，这就足够了，可以在Hibernate中使用大对象类型。

4.10. 大对象（LOBs）

JPA 提供了 `@Lob` 注解，用于指定字段应该被持久化为 BLOB 或 CLOB。

`@Lob` 注解的语义 规范实际上说，该字段应该被持久化为

...作为一个数据库支持的大对象类型。

这是相当不清晰的，而且规范还继续说

...Lob 注解的处理是提供者相关的...

这并没有太大帮助。

Hibernate 对这个注解的解释是我们认为最合理的方式。在 Hibernate 中，一个被 `@Lob` 注解的属性将使用 PreparedStatement 的 `setClob()` 或 `setBlob()` 方法写入到 JDBC，并且将使用 ResultSet 的 `getClob()` 或 `getBlob()` 方法从 JDBC 中读取。

但是，通常情况下使用这些 JDBC 方法是不必要的！JDBC 驱动完全有能力在 String 和 CLOB 之间，或者在 byte[] 和 BLOB 之间进行转换。所以，除非你明确需要使用这些 JDBC LOB API，否则你不需要 `@Lob` 注解。

相反，正如我们在“列长度和自适应列类型”中看到的那样，你只需要指定足够大的列长度来容纳你计划写入该列的数据。

不幸的是，PostgreSQL 的驱动程序不允许通过 JDBC LOB API 读取 BYTEA 或 TEXT 类型的列。

这个 Postgres 驱动程序的限制导致了一个整个博客界和 stackoverflow 上的问题回答者推荐使用复杂的方式来修改用于 Postgres 的 Hibernate 方言，以允许使用 `setString()` 写入属性，然后使用 `getString()` 读取。

但是，简单地移除 `@Lob` 注解会达到完全相同的效果。

总结：

- 在 PostgreSQL 中，`@Lob` 总是表示 OID 类型，
- `@Lob` 永远不应该用于映射 BYTEA 或 TEXT 类型的列，而

- 请不要相信你在 stackoverflow 上看到的一切。

最后，作为一种替代方法，Hibernate 允许你声明一个类型为 `java.sql.Blob` 或 `java.sql.Clob` 的属性。

```
@Entity
class Book {
    ...
    Clob text;
    Blob coverArt;
    ....
}
```

优点是 `java.sql.Clob` 或 `java.sql.Blob` 在原则上可以索引多达 2^63 个字符或字节的数据，比你可以放入 Java 字符串或 byte[] 数组（或你的计算机）中的数据要多得多。

要给这些字段分配一个值，我们需要使用 `LobHelper`。我们可以从 Session 中获取它：

```
LobHelper helper = session.getLobHelper();
book.text = helper.createClob(text);
book.coverArt = helper.createBlob(image);
```

原则上，Blob 和 Clob 对象提供了从服务器读取或流式传输 LOB 数据的有效方法。

```
Book book = session.find(Book.class, bookId);
String text = book.text.getSubString(1, textLength);
InputStream bytes = book.images.getBinaryStream();
```

当然，这里的行为非常依赖于 JDBC 驱动程序，所以我们不能保证在你的数据库上这样做是明智的。

4.11. 映射可嵌入类型到 UDTs 或 JSON

有几种可用的方式可以在数据库端表示可嵌入类型。

可嵌入类型作为 UDTs

首先，一个非常好的选择，至少在 Java 记录类型的情况下，并且对于支持用户定义类型（UDTs）的数据库来说，是定义一个代表记录类型的 UDT。Hibernate 6 使得这个过程非常简单。只需使用新的 `@Struct` 注解为记录类型或持有对它的引用的属性进行注解：

```
@Embeddable
@Struct(name="PersonName")
record Name(String firstName, String middleName, String lastName) {}
@Entity
class Person {
    ...
    Name name;
    ...
}
```

这将得到以下的 UDT：

```
create type PersonName as (firstName varchar(255), middleName varchar(255), lastName varchar(255))
```

并且 Author 表的 name 列将具有类型 PersonName。

将可嵌入类型映射到 JSON

第二个可用的选项是将可嵌入类型映射到 JSON（或 JSONB）列。现在，如果你是从零开始定义数据模型，我们不会完全推荐这样做，但如果你需要映射具有预定义 JSON 类型列的现有表格，这是一个有用的方式。由于可嵌入类型是可以嵌套的，我们可以使用这种方式映射某些 JSON 格式，甚至可以使用 HQL 查询 JSON 属性。

目前，不支持 JSON 数组！

要将可嵌入类型的属性映射到 JSON，我们必须为属性注解 `@JdbcTypeCode(SqlTypes.JSON)`，而不是为可嵌入类型进行注解。但是，如果我们想要使用 HQL 查询它的属性，可嵌入类型 Name 仍然应该被注解为 `@Embeddable`。

```
@Embeddable
record Name(String firstName, String middleName, String lastName) {}
@Entity
class Person {
    ...
    @JdbcTypeCode(SqlTypes.JSON)
    Name name;
    ...
}
```

我们还需要将 Jackson 或 JSONB 的实现（例如 Yasson）添加到运行时类路径中。如果我们想使用 Jackson，我们可以在 Gradle 构建中添加以下行：

```
runtimeOnly 'com.fasterxml.jackson.core:jackson-databind:{jacksonVersion}'
```

现在 Author 表的 name 列将具有类型 jsonb，Hibernate 将自动使用 Jackson 将 Name 序列化为 JSON 格式，并从 JSON 格式反序列化为 Name。

4.12. SQL列类型映射总结

正如我们所见，有很多注解会影响Java类型在DDL中映射到SQL列类型。在这里，我们总结了在本章的后半部分中我们刚刚看到的这些注解，以及在之前章节中已经提到的一些注解。

映射SQL列类型的注解

注解	解释
`@Enumerated`	指定枚举类型应该如何持久化。
`@Nationalized`	使用国际化字符类型：NCHAR、NVARCHAR 或 NCLOB。
`@Lob`	使用 JDBC LOB APIs 来读取和写入被注解的属性。
`@Array`	将集合映射到具有指定长度的 SQL ARRAY 类型。
`@Struct`	将可嵌入类型映射到具有给定名称的 SQL UDT。
`@TimeZoneStorage`	指定时区信息应该如何持久化。
`@JdbcType` 或 `@JdbcTypeCode`	使用 JdbcType 的实现来映射任意的 SQL 类型。

此外，还有一些配置属性对基本类型如何映射到SQL列类型具有全局影响：

类型映射设置

配置属性名	目的
`hibernate.use_nationalized_character_data`	启用默认情况下使用国际化字符类型。
`hibernate.type.preferred_boolean_jdbc_type`	指定映射布尔类型的默认SQL列类型。
`hibernate.type.preferred_uuid_jdbc_type`	指定映射UUID的默认SQL列类型。
`hibernate.type.preferred_duration_jdbc_type`	指定映射Duration的默认SQL列类型。
`hibernate.type.preferred_instant_jdbc_type`	指定映射Instant的默认SQL列类型。
`hibernate.timezone.default_storage`	指定存储时区信息的默认策略。

这些是全局设置，因此相当笨拙。我们建议除非你有一个非常充分的理由，否则不要去修改这些设置。

在这个章节中，我们还有一个主题想要讨论。

4.13. 映射到公式

Hibernate 允许我们将实体的属性映射到涉及到映射表的列的 SQL 公式。因此，该属性是一种"派生"值。

映射到公式的注解

注解	目的
`@Formula`	将属性映射到 SQL 公式。
`@JoinFormula`	将关联映射到 SQL 公式。
`@DiscriminatorFormula`	在单表继承中使用 SQL 公式作为鉴别器。

例如：

```
@Entity
class Order {
    ...
    @Column(name = "sub_total", scale=2, precision=8)
    BigDecimal subTotal;

    @Column(name = "tax", scale=4, precision=4)
    BigDecimal taxRate;

    @Formula("sub_total * (1.0 + tax)")
    BigDecimal totalWithTax;
    ...
}
```

在这个例子中，`totalWithTax` 属性通过 SQL 公式 `sub_total * (1.0 + tax)` 计算得出。

4.14. 派生标识

当一个实体的部分主键从关联的“父”实体继承而来时，该实体具有派生标识。我们在讨论具有共享主键的一对一关联时已经遇到了一种派生标识的特殊情况。

但是，@ManyToOne 关联也可以是派生标识的一部分。也就是说，可以将外键列或多个外键列包含为组合主键的一部分。在Java方面，有三种不同的方式来表示这种情况：

1. 使用带有@IdClass但不带有@MapsId的方式。
2. 使用带有@IdClass和@MapsId的方式。
3. 使用带有@EmbeddedId和@MapsId的方式。

假设我们有一个如下定义的Parent实体类：

```
@Entity
class Parent {
    @Id
    Long parentId;

    ...
}
```

parentId字段保存了Parent表的主键，该主键也将成为每个Child实体的复合主键的一部分。

第一种方式

在第一种稍微简单的方法中，我们定义了一个@IdClass来表示Child的主键：

```
class DerivedId {
    Long parent;
    String childId;

    // 构造函数，equals，hashcode等
    ...
}
```

并且Child实体类上使用了@Id注解的@ManyToOne关联：

```
@Entity
@IdClass(DerivedId.class)
class Child {
    @Id
    String childId;

    @Id @ManyToOne
    @JoinColumn(name="parentId")
    Parent parent;

    ...
}
```

然后，Child表的主键由列(childId,parentId)组成。

第二种方式

这种方法是可以的，但有时最好为主键的每个元素拥有一个字段。我们可以使用我们之前遇到的@MapsId注解：

```
@Entity
@IdClass(DerivedId.class)
class Child {
    @Id
    Long parentId;
    @Id
    String childId;

    @ManyToOne
    @MapsId(Child_.PARENT_ID) // 对Child.parentId进行类型安全引用
    @JoinColumn(name="parentId")
    Parent parent;

    ...
}
```

我们使用了之前看到的方法来以类型安全的方式引用Child的parentId属性。

请注意，我们必须将列映射信息放在@MapsId注解上，而不是@Id字段上。

我们必须稍微修改我们的@IdClass，以便字段名称对齐：

```
class DerivedId {
    Long parentId;
    String childId;

    // 构造函数，equals，hashcode等
    ...
}
```

第三种方式

第三种选择是将我们的@IdClass重新定义为@Embeddable。实际上，我们不需要更改DerivedId类，但是我们需要添加该注解。

```
@Embeddable
class DerivedId {
    Long parentId;
    String childId;

    // 构造函数，equals，hashcode等
    ...
}
```

然后我们可以在Child中使用@EmbeddedId：

```
@Entity
class Child {
    @EmbeddedId
    DerivedId id;

    @ManyToOne
    @MapsId(DerivedId_.PARENT_ID) // 对DerivedId.parentId进行类型安全引用
    @JoinColumn(name="parentId")
    Parent parent;

    ...
}
```

在@IdClass和@EmbeddedId之间的选择取决于个人喜好。@EmbeddedId或许更加DRY（Don't Repeat Yourself）。

4.15. 添加约束

数据库约束非常重要。即使你确信你的程序没有bug 😜，它可能并不是唯一一个能够访问数据库的程序。约束有助于确保不同的程序（以及人类管理员）可以友好地相互配合。

Hibernate会自动将某些约束添加到生成的DDL中：主键约束、外键约束和一些唯一约束。但通常需要：

- 添加额外的唯一约束，
- 添加检查约束，或者
- 自定义外键约束的名称。

我们已经学过如何使用 @ForeignKey 注解来指定外键约束的名称。

有两种方法可以向表中添加唯一约束：

1. 使用 `@Column(unique=true)` 指定单列唯一键，或者
2. 使用 `@UniqueConstraint` 注解在多列上定义唯一性约束。

```
@Entity
@Table(uniqueConstraints=@UniqueConstraint(columnNames={"title", "year", "publisher_id"}))
class Book { ... }
```

这个注解可能看起来有点丑，但实际上它作为文档是非常有用的。

`@Check` 注解可以向表中添加检查约束。

```
@Entity
@Check(name="ValidISBN", constraints="length(isbn)=13")
class Book { ... }
```

`@Check` 注解通常用在字段级别上：

```
@Id
@Check(constraints="length(isbn)=13")
String isbn;
```


Hibernate 6 中文文档（五）

目录

5.与数据库交互

5.1 持久化上下文

5.2 创建会话

5.3 管理事务

5.4. 持久化上下文的操作

5.5. 级联持久化操作

5.6. 代理和延迟加载

5.7. 实体图和急加载

5.8. 刷新会话

5.9. 查询

5.10. HQL查询

5.11. Criteria查询

5.12. 更舒适的编写Criteria查询的方式

5.13. 原生SQL查询

5.14. 查询结果的限制、分页和排序

5.15. 表达投影列表

5.16. 命名查询

5.17. 控制按ID查找

5.18. 直接与JDBC交互

5.19. 当事情出错时该怎么办

5.与数据库交互

要与数据库交互，也就是执行查询，或者插入、更新、删除数据，我们需要一个以下对象的实例：

- 一个 JPA EntityManager，
- 一个 Hibernate Session，
- 或者一个 Hibernate StatelessSession。

Session 接口扩展了 EntityManager，所以这两个接口之间唯一的区别在于 Session 提供了一些额外的操作。

实际上，在 Hibernate 中，每个 EntityManager 都是一个 Session，你可以像这样缩小范围：

```
Session session = entityManager.unwrap(Session.class);
```

Session（或者 EntityManager）的实例是有状态的会话。它通过持久化上下文上的操作来调解你的程序与数据库之间的交互。

在本章中，我们不会详细讨论 StatelessSession。当我们谈论性能时，我们会回到这个非常有用的 API。你现在需要知道的是，无状态会话没有持久化上下文。

尽管如此，我们还是应该让你知道，有些人更喜欢在任何地方使用 StatelessSession。它是一个更简单的编程模型，并且允许开发人员更直接地与数据库交互。

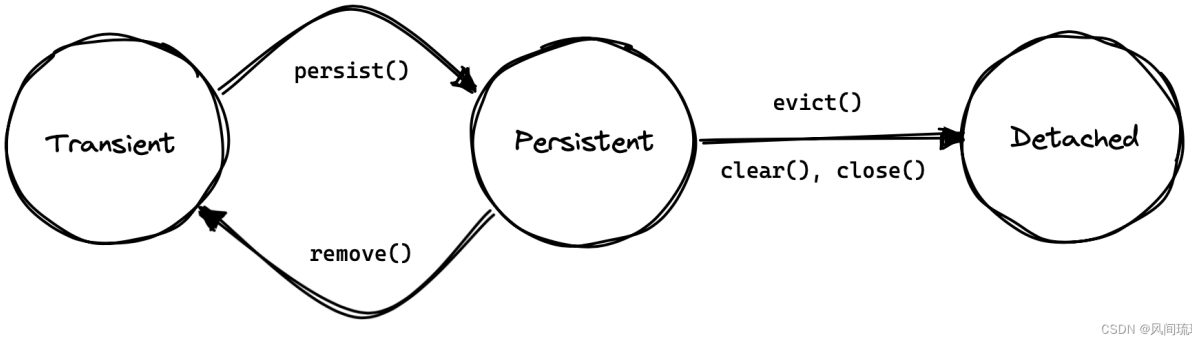
有状态会话当然有它们的优势，但是更难以理解，当出现问题时，错误消息可能更难以理解。

5.1 持久化上下文

持久化上下文是一种缓存，有时我们称之为“一级缓存”，以区别于二级缓存。在持久化上下文的范围内，每个从数据库中读取的实体实例，以及在持久化上下文的范围内使实体持久化的每个新实体，上下文都会保持从实体实例的标识符到实例本身的独特映射。

因此，一个实体实例在给定的持久化上下文中可能处于以下三种状态之一：

- **瞬时态（transient）** — 从未持久化，也不与持久化上下文关联，
- **持久态（persistent）** — 目前与持久化上下文关联，
- **游离态（detached）** — 以前在另一个会话中持久化，但目前不与这个持久化上下文关联。



实体生命周期 在任何给定时刻，一个实例最多只能与一个持久化上下文关联。

持久化上下文的生命周期通常对应于一个事务的生命周期，尽管可能存在一个持久化上下文跨越多个数据库级事务的情况，这些事务形成一个单一的逻辑工作单元。

> 持久化上下文，即 Session 或 EntityManager，绝对不能在多个线程之间或在并发事务之间共享。

> 如果你不小心在不同线程之间泄漏了一个会话，会导致严重问题。

> **容器管理的持久化上下文**
>
> 在容器环境中，通常会为你管理事务范围内的持久化上下文的生命周期。

我们喜欢持久化上下文有几个原因。

- 它们有助于避免数据别名问题：如果我们在代码的某个部分修改了一个实体，那么在同一持久化上下文内执行的其他代码将看到我们的修改。
- 它们实现了自动脏数据检查：在修改实体后，我们不需要执行任何显式操作来通知 Hibernate 将修改传播回数据库。相反，当会话被刷新时，这些更改将自动与数据库同步。
- 当在给定的工作单元中多次请求给定实体实例时，它们可以通过避免访问数据库来提高性能。
- 它们使得能够透明地批处理多个数据库操作。

持久化上下文还允许我们在执行实体图操作时检测循环引用。（即使在无状态会话中，执行查询时，我们也需要某种临时缓存已访问的实体实例。）

然而，有状态会话带有一些非常重要的限制，因为：

- 持久化上下文不是线程安全的，不能在多个线程之间共享，
- 持久化上下文不能在不相关的事务之间重用，因为那会破坏事务的隔离性和原子性。

此外，持久化上下文保持对其所有实体的硬引用，阻止它们被垃圾回收。因此，会话必须在工作单元完成后被丢弃。

> 如果你不完全理解前述内容，请返回并重新阅读，直到理解为止。许多人由于错误管理 Hibernate 会话或 JPA EntityManager 的生命周期而遭受了很多痛苦。

最后，需要指出的是，持久化上下文对于给定工作单元的性能是帮助还是妨碍，取决于工作单元的性质。因此，Hibernate 提供了有状态和无状态的会话。

5.2 创建会话

如果我们坚持使用标准的 JPA 定义的 API，我们可以看到在配置中如何使用 JPA XML 获取 EntityManagerFactory。毫不奇怪，我们可以使用这个对象来创建 EntityManager：

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
```

当我们使用完 EntityManager 后，应该显式关闭它：

```
entityManager.close();
```

另一方面，如果我们是从一个 SessionFactory 开始，就像在配置中使用 Hibernate API 那样，我们可以使用：

```
Session session = sessionFactory.openSession();
```

但是同样需要在使用完之后进行清理：

```
session.close();
```

注入 EntityManager

如果你在某种容器环境下编写代码，你可能会通过某种依赖注入方式获取 EntityManager。例如，在 Java（或 Jakarta）EE 中，你可以这样写：

```
@PersistenceContextEntityManager entityManager;
```

在 Quarkus 中，注入是由 CDI 处理的：

```
@InjectEntityManager entityManager;
```

在非容器环境下，我们还需要编写代码来管理数据库事务。

5.3 管理事务

使用 JPA 标准的 API，EntityTransaction 接口允许我们控制数据库事务。我们推荐的惯用法如下：

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
EntityTransaction tx = entityManager.getTransaction();
try {
    tx.begin();
    // 进行一些操作
    // ...
    tx.commit();
} catch (Exception e) {
    if (tx.isActive()) tx.rollback();
    throw e;
} finally {
    entityManager.close();
}
```

使用 Hibernate 的原生 API，我们可能会编写非常相似的代码，但由于这种代码非常繁琐，我们有一个更好的选择：

```
sessionFactory.inTransaction(session -> {    // 进行操作    // ...});
```

> **容器管理的事务**
>
> 在容器环境中，通常由容器本身负责管理事务。在 Java EE 或 Quarkus 中，你可能会使用 @Transactional 注解来指定事务的边界。

5.4. 持久化上下文的操作

当然，我们需要一个实体管理器（EntityManager）的主要原因是要对数据库进行操作。以下是让我们与持久化上下文（persistence context）进行交互并安排对数据进行修改的重要操作：

表格 33. 修改数据和管理持久化上下文的方法

方法名及参数	作用
persist(Object)	将一个瞬时对象变为持久对象，并安排将SQL插入语句延后执行
remove(Object)	将一个持久对象变为瞬时对象，并安排将SQL删除语句延后执行
merge(Object)	将给定的游离对象的状态复制到相应的托管持久对象上，并返回该持久对象
detach(Object)	将持久对象与当前会话（session）分离，不影响数据库
clear()	清空持久化上下文并分离其中的所有实体
flush()	检测对持久对象进行的更改，将数据库状态与会话状态同步，执行SQL插入、更新和删除语句

请注意，persist() 和 remove() 对数据库没有立即影响，只是安排了一个命令以便稍后执行。还要注意，对于有状态的会话，没有 update() 操作。当会话刷新时，修改将被自动检测到。

另一方面，除了 getReference()，以下操作都会立即访问数据库：

表格 34. 读取和锁定数据的方法

方法名及参数	作用
find(Class,Object)	根据类型和id获取持久对象
find(Class,Object,LockModeType)	根据类型和id获取持久对象，请求给定的乐观或悲观锁定模式
getReference(Class,id)	获取持久对象的引用，不实际从数据库加载其状态
getReference(Object)	获取与给定游离实例具有相同标识的持久对象的引用，不实际从数据库加载其状态
refresh(Object)	使用新的SQL查询刷新对象的持久状态，以从数据库检索其当前状态
refresh(Object,LockModeType)	使用新的SQL查询刷新对象的持久状态，以从数据库检索其当前状态，并请求给定的乐观或悲观锁定模式
lock(Object, LockModeType)	对持久对象进行乐观或悲观锁定

这些操作中的任何一个都可能引发异常。如果在与数据库交互时发生异常，没有很好的方法来将当前持久化上下文的状态与数据库表中的状态重新同步。

因此，在任何方法引发异常后，会话被视为不可用。

> 持久化上下文是脆弱的。如果从Hibernate收到异常，应立即关闭并丢弃当前会话。如果需要，可以打开一个新的会话，但是首先应该将不好的会话丢弃掉。

到目前为止，我们看到的每个操作都影响作为参数传递的单个实体实例。但是，有一种方法可以设置事务，使操作会传播到相关的实体。

5.5. 级联持久化操作

在很多情况下，子实体的生命周期完全依赖于某个父实体的生命周期。这在多对一和一对一关联中特别常见，尽管在多对多关联中很少见。

例如，在同一个事务中将订单（Order）及其所有商品（Items）都设置为持久状态是相当常见的，或者一次性删除项目（Project）及其文件（Files）也是如此。这种关系有时被称为整体/部分关系（whole/part-type relationship）。

级联（Cascading）是一种便利的功能，它允许我们将持久化上下文的操作之一从父实体传播到其子实体。为了设置级联，我们需要在关联映射注解中的某一个（通常是 @OneToMany 或 @OneToOne）中指定 cascade 成员。

```
@Entity
class Order {
    ...
    @OneToMany(mappedBy = Item_.ORDER,
               // 级联 persist()、remove() 和 refresh() 从 Order 到 Item
               cascade = {PERSIST, REMOVE, REFRESH},
               // 如果Item从其父订单的项目集合中移除，也删除它
               orphanRemoval = true)
    private Set<Item> items;
    ...
}
```

orphanRemoval 指示如果一个 Item 从其父订单的项目集合中移除，它应该被自动删除。

5.6. 代理和延迟加载

我们的数据模型是一组相互关联的实体，在Java中，整个数据集将被表示为一个庞大的相互关联的对象图。这个图可能是断开连接的，但更可能是连接的，或者由相对较小数量的连接子图组成。

因此，当我们从数据库中检索属于此图的对象并在内存中实例化它时，我们不能递归地检索和实例化所有相关的实体。这不仅会在VM端浪费内存，而且这个过程将涉及大量的数据库服务器往返，或者涉及大规模的表的多维笛卡尔积，或者两者兼而有之。因此，我们被迫在某个地方截断图。

Hibernate使用代理和延迟加载来解决这个问题。代理是一个伪装成真实实体或集合的对象，但实际上不包含任何状态，因为该状态尚未从数据库中获取。当你调用代理的方法时，Hibernate将检测到调用并在允许调用传递到真实实体对象或集合之前，从数据库中获取状态。

现在来谈谈需要注意的地方：

Hibernate仅对当前与持久化上下文关联的实体执行此操作。一旦会话结束，并且持久化上下文被清理，代理将无法再次获取，代替它的方法将抛出令人讨厌的 LazyInitializationException 异常。

为了访问单个实体实例的状态，通过数据库进行一次往返是访问数据的最低效方式。这几乎不可避免地导致了我们将在稍后讨论的臭名昭著的 N+1 查询问题。

> 我们稍微超前了一点，但让我们快速提到我们推荐的一般策略，以规避这些注意事项：
>
> 1. 所有关联应该设置为 fetch=LAZY，以避免在不需要数据时获取额外的数据。正如我们前面提到的，这个设置不是 @ManyToOne 关联的默认值，必须显式指定。
> 2. 但是，努力避免编写会触发延迟加载的代码。相反，在工作单元的开始时，使用 Association fetching 中描述的技术之一提前获取所有需要的数据，通常使用 HQL 中的 join fetch 或者 EntityGraph。

需要注意的是，有些操作可能会在未获取代理的状态的情况下执行。首先，我们总是可以获取其标识符：

```
var pubId = entityManager.find(Book.class, bookId).getPublisher().getId(); // 不会获取发布商
```

其次，我们可以创建一个与代理关联的关联：

```
book.setPublisher(entityManager.getReference(Publisher.class, pubId)); // 不会获取发布商
```

有时，测试代理或集合是否已从数据库获取是很有用的。JPA 允许我们使用 PersistenceUnitUtil 来实现这一点：

```
boolean authorsFetched = entityManagerFactory.getPersistenceUnitUtil().isLoaded(book.getAuthors());
```

Hibernate 提供了一个稍微简单的方法：

```
boolean authorsFetched = Hibernate.isInitialized(book.getAuthors());
```

但 Hibernate 类的静态方法让我们可以做更多的事情，熟悉这些方法是值得的。

其中特别有趣的是，这些操作使我们能够在未获取集合的状态的情况下处理未获取的集合。例如，考虑以下代码：

```
Book book = session.find(Book.class, bookId);  // 仅获取 Book，将作者保持未获取状态
Author authorRef = session.getReference(Author.class, authorId);  // 获取一个未获取的代理
boolean isByAuthor = Hibernate.contains(book.getAuthors(), authorRef); // 不会获取数据
```

此代码片段保持了集合 `book.authors` 和代理 `authorRef` 的未获取状态。

最后，Hibernate.initialize() 是一个方便的方法，用于强制获取代理或集合的数据：

```
Book book = session.find(Book.class, bookId);  // 仅获取 Book，将作者保持未获取状态
Hibernate.initialize(book.getAuthors());  // 获取 Authors 的数据
```

但是，这段代码非常低效，需要两次往返数据库，而原则上可以使用一次查询就获取到数据。

从上述讨论中可以看出，我们需要一种方法来请求使用数据库连接来急加载关联，以避免臭名昭著的 N+1 查询。其中一种方法是通过将 EntityGraph 传递给 find() 方法来实现。

5.7. 实体图和急加载

当一个关联被映射为 `fetch=LAZY` 时，在调用 `find()` 方法时，默认情况下不会被立即获取。我们可以通过将 `EntityGraph` 传递给 `find()` 方法来请求急加载（立即加载）关联。

标准的JPA API有点笨重：

```
var graph = entityManager.createEntityGraph(Book.class);
graph.addSubgraph(Book_.publisher);
Book book = entityManager.find(Book.class, bookId, Map.of(SpecHints.HINT_SPEC_FETCH_GRAPH, graph));
```

这种方式不是类型安全的，而且冗长。Hibernate有一个更好的方法：

```
var graph = session.createEntityGraph(Book.class);
graph.addSubgraph(Book_.publisher);
Book book = session.byId(Book.class).withFetchGraph(graph).load(bookId);
```

这段代码向我们的SQL查询添加了一个左外连接，一起获取了关联的发布商（Publisher）和书籍（Book）。

我们甚至可以将额外的节点附加到我们的 `EntityGraph` 上：

```
var graph = session.createEntityGraph(Book.class);
graph.addSubgraph(Book_.publisher);
graph.addPluralSubgraph(Book_.authors).addSubgraph(Author_.person);
Book book = session.byId(Book.class).withFetchGraph(graph).load(bookId);
```

这将导致一个包含四个左外连接的SQL查询。

> 在上面的代码示例中，类 `Book_` 和 `Author_` 是我们之前见过的JPA元模型生成器生成的。它们允许我们以完全类型安全的方式引用我们模型的属性。在下面讨论标准查询时，我们将再次使用它们。

JPA规定，任何给定的 `EntityGraph` 可以以两种不同的方式进行解释：

- 一个 `fetch` 图（fetch graph）明确指定了应该被急加载的关联。任何不属于实体图的关联将被代理，并且只在需要时懒加载。
- 一个 `load` 图（load graph）指定实体图中的关联应该在除了映射为 `fetch=EAGER` 的关联之外额外加载。

你是对的，这些名字没有意义。但是不用担心，如果按照我们的建议，将你的关联映射为 `fetch=LAZY`，那么 "fetch" 图 和 "load" 图之间没有区别，所以名字并不重要。

> JPA甚至规定了使用注解定义命名实体图的方式。但是基于注解的API非常冗长，所以使用起来并不值得。

5.8. 刷新会话

不时地，会触发刷新操作，会话将内存中的脏状态（即，与持久化上下文关联的实体状态的修改）与数据库中的持久状态同步。当然，它通过执行SQL的INSERT、UPDATE和DELETE语句来实现这一点。

默认情况下，会触发刷新操作的时机包括：

- 当当前事务提交时，例如，调用 `Transaction.commit()` 方法时。
- 在执行查询，其结果可能受到内存中脏状态同步影响的查询之前。
- 当程序直接调用 `flush()` 方法时。

请注意，通常不会在 `Session` 接口的方法（如 `persist()` 和 `remove()`）中同步执行SQL语句。如果需要同步执行SQL，可以使用 `StatelessSession`。

这种行为可以通过显式设置刷新模式来控制。例如，要禁用在查询执行之前触发的刷新操作，可以调用：

```
entityManager.setFlushMode(FlushModeType.COMMIT);
```

Hibernate允许比JPA更精细地控制刷新模式：

```
session.setHibernateFlushMode(FlushMode.MANUAL);
```

由于刷新是一个相对昂贵的操作（会话必须对持久化上下文中的每个实体进行脏检查），将刷新模式设置为 `COMMIT` 有时可能是一种有用的优化。

表格 35. 刷新模式

Hibernate 刷新模式	JPA FlushModeType	解释
MANUAL	NEVER	从不自动刷新
COMMIT	COMMIT	在事务提交前刷新
AUTO	AUTO	在事务提交前和可能受到内存中修改影响的查询执行之前刷新
ALWAYS	AUTO	在事务提交前和每次查询执行之前刷新

减少刷新成本的另一种方式是以只读模式加载实体：

- `Session.setDefaultReadOnly(false)` 指定由给定会话加载的所有实体应该默认以只读模式加载。
- `SelectionQuery.setReadOnly(false)` 指定给定查询返回的每个实体应该以只读模式加载。
- `Session.setReadOnly(Object, false)` 指定会话中已加载的给定实体应该切换到只读模式。

在只读模式下，不需要对实体实例进行脏检查。

5.9. 查询

Hibernate提供了三种互补的查询方法：

- **Hibernate查询语言（HQL）**：这是JPQL的一个极其强大的超集，它抽象了大多数现代SQL方言的特性。
- **JPA标准的Criteria查询API**：此外，还有一些扩展，允许通过类型安全的API以编程方式构建几乎任何HQL查询。
- **原生SQL查询**：当其他方法无法满足需求时，可以使用原生SQL查询。

5.10. HQL查询

对查询语言的全面讨论需要的文字量几乎和本文其余部分一样多。幸运的是，HQL已经在《Hibernate查询语言指南》中得到详细（也是详尽）的描述。在这里重复这些信息并没有意义。

在这里，我们想看看如何通过Session或EntityManager API执行查询。我们调用的方法取决于查询的类型：

- **选择查询（Selection queries）**：返回结果列表，但不修改数据。
- **变更查询（Mutation queries）**：修改数据，并返回修改的行数。

选择查询通常以关键字 `select` 或 `from` 开始，而变更查询则以关键字 `insert`、`update` 或 `delete` 开始。

表格 36. 执行HQL

类型	Session方法	EntityManager方法	查询执行方法
选择查询	`createSelectionQuery(String, Class)`	`createQuery(String, Class)`	`getResultList()`, `getSingleResult()` 或 `getSingleResultOrNull()`
变更查询	`createMutationQuery(String)`	`createQuery(String)`	`executeUpdate()`

因此，对于Session API，我们可以编写如下代码：

```
List<Book> matchingBooks =
        session.createSelectionQuery("from Book where title like :titleSearchPattern", Book.class)
            .setParameter("titleSearchPattern", titleSearchPattern)
            .getResultList();
```

或者，如果我们坚持使用JPA标准的API：

```
List<Book> matchingBooks =
        entityManager.createQuery("select b from Book b where b.title like :titleSearchPattern", Book.class)
            .setParameter("titleSearchPattern", titleSearchPattern)
            .getResultList();
```

`createSelectionQuery()` 和 `createQuery()` 唯一的区别在于，如果传递了 `insert`、`delete` 或 `update`，`createSelectionQuery()` 会抛出异常。

在上述查询中，`:titleSearchPattern` 被称为命名参数。我们也可以用数字来标识参数，这种方式称为序数参数。

```
List<Book> matchingBooks =
        session.createSelectionQuery("from Book where title like ?1", Book.class)
            .setParameter(1, titleSearchPattern)
            .getResultList();
```

当查询有多个参数时，命名参数通常更易读，即使稍微冗长。

永远不要将用户输入与HQL连接起来，然后将连接后的字符串传递给 `createSelectionQuery()`。这样做会使攻击者有可能在数据库服务器上执行任意代码。

如果我们期望查询返回单个结果，可以使用 `getSingleResult()`。

```
Book book =
        session.createSelectionQuery("from Book where isbn = ?1", Book.class)
            .setParameter(1, isbn)
            .getSingleResult();
```

或者，如果我们期望它最多返回一个结果，可以使用 `getSingleResultOrNull()`。

```
Book bookOrNull =
        session.createSelectionQuery("from Book where isbn = ?1", Book.class)
            .setParameter(1, isbn)
            .getSingleResultOrNull();
```

当然，差异在于，如果数据库中没有匹配的行，`getSingleResult()` 会抛出异常，而 `getSingleResultOrNull()` 只是返回 `null`。

默认情况下，Hibernate在执行查询之前会对持久化上下文中的实体进行脏检查，以确定是否应该刷新会话。如果与持久化上下文关联的实体很多，这可能是一个昂贵的操作。

要禁用此行为，可以将刷新模式设置为 `COMMIT` 或 `MANUAL`：

```
Book bookOrNull =
        session.createSelectionQuery("from Book where isbn = ?1", Book.class)
            .setParameter(1, isbn)
            .setHibernateFlushMode(MANUAL)
            .getSingleResult();
```

将刷新模式设置为 `COMMIT` 或 `MANUAL` 可能会导致查询返回过时的结果。

有时，我们需要在运行时构建一个查询，根据一组可选条件。为此，JPA提供了一个API，允许以编程方式构建查询。

5.11. Criteria查询

假设我们正在实现一个搜索屏幕，在这个屏幕上，系统用户可以通过多种不同的方式来限制查询结果集。例如，我们可能允许他们按书名和/或作者名搜索书籍。当然，我们可以通过字符串连接构造一个HQL查询，但这样有点脆弱，因此有一个替代方法会很好。

HQL是基于Criteria对象实现的。

实际上，在Hibernate 6中，每个HQL查询在被转换为SQL之前都会被编译为一个Criteria查询。这确保了HQL和Criteria查询的语义是相同的。

首先，我们需要一个用于构建Criteria查询的对象。使用JPA标准API，这将是一个CriteriaBuilder，我们可以从EntityManagerFactory中获取：

```
CriteriaBuilder builder = entityManagerFactory.getCriteriaBuilder();
```

但是如果我们有一个SessionFactory，我们可以得到一个更好的东西，一个HibernateCriteriaBuilder：

```
HibernateCriteriaBuilder builder = sessionFactory.getCriteriaBuilder();
```

HibernateCriteriaBuilder扩展了CriteriaBuilder，并添加了许多JPQL没有的操作。

> 如果你使用的是EntityManagerFactory，请不要绝望，你有两种很好的方法可以获取与该工厂关联的HibernateCriteriaBuilder。要么：

```
HibernateCriteriaBuilder builder =
 entityManagerFactory.unwrap(SessionFactory.class).getCriteriaBuilder();
```

或者简单地：

```
HibernateCriteriaBuilder builder =        (HibernateCriteriaBuilder)
entityManagerFactory.getCriteriaBuilder();
```

现在我们可以创建一个Criteria查询了。

```
CriteriaQuery<Book> query = builder.createQuery(Book.class);
Root<Book> book = query.from(Book.class);
Predicate where = builder.conjunction();
if (titlePattern != null) {
    where = builder.and(where, builder.like(book.get(Book_.title), titlePattern));
}
if (namePattern != null) {
    Join<Book,Author> author = book.join(Book_.author);
    where = builder.and(where, builder.like(author.get(Author_.name), namePattern));
}
query.select(book).where(where)
    .orderBy(builder.asc(book.get(Book_.title)));
```

在这里，与以前一样，类 `Book_` 和 `Author_` 是由Hibernate的JPA元模型生成器生成的。

> 请注意，我们没有把 `titlePattern` 和 `namePattern` 当作参数来处理。这是安全的，因为默认情况下，Hibernate会自动透明地将传递给CriteriaBuilder的字符串视为JDBC参数。

Criteria查询的执行方式几乎与执行HQL相同。

表格 37. 执行Criteria查询

类型	Session方法	EntityManager方法	查询执行方法
选择查询	`createSelectionQuery(CriteriaQuery)`	`createQuery(CriteriaQuery)`	`getResultList()`, `getSingleResult()` 或 `getSingleResultOrNull()`
变更查询	`createMutationQuery(CriteriaUpdate)` or `createQuery(CriteriaDelete)`	`createQuery(CriteriaUpdate)` or `createQuery(CriteriaDelete)`	`executeUpdate()`

例如：

```
List<Book> matchingBooks =        session.createSelectionQuery(query)            .getResultList();
```

更新（Update）、插入（Insert）和删除（Delete）查询的工作方式类似：

```
CriteriaDelete<Book> delete = builder.createCriteriaDelete(Book.class);
Root<Book> book = delete.from(Book.class);
delete.where(builder.lt(builder.year(book.get(Book_.publicationDate)), 2000));
session.createMutationQuery(delete).executeUpdate();
```

> 甚至可以将HQL查询字符串转换为Criteria查询，并在执行之前以编程方式修改查询：
>
> ```
> HibernateCriteriaBuilder builder = sessionFactory.getCriteriaBuilder();
> var query = builder.createQuery("from Book where year(publicationDate) > 2000", Book.class);
> var root = (Root<Book>) query.getRootList().get(0);
> query.where(builder.like(root.get(Book_.title), builder.literal("Hibernate%")));
> query.orderBy(builder.asc(root.get(Book_.title)), builder.desc(root.get(Book_.isbn)));
> List<Book> matchingBooks = session.createSelectionQuery(query).getResultList();
> ```

你是否觉得上面的代码有点太冗长了？我们也有同感。

5.12. 更舒适的编写Criteria查询的方式

实际上，使得JPA标准的Criteria API不够符合人体工学的原因是，需要将所有CriteriaBuilder的操作作为实例方法调用，而不是将它们作为静态函数。之所以是这样，是因为每个JPA提供商都有自己的CriteriaBuilder实现。

Hibernate 6.3引入了助手类CriteriaDefinition，以减少编写criteria查询时的冗长。我们的示例代码如下：

```
CriteriaQuery<Book> query =
        new CriteriaDefinition(entityManagerFactory, Book.class) {{
            select(book);
            if (titlePattern != null) {
                restrict(like(book.get(Book_.title), titlePattern));
            }
            if (namePattern != null) {
                var author = book.join(Book_.author);
                restrict(like(author.get(Author_.name), namePattern));
            }
            orderBy(asc(book.get(Book_.title)));
        }};
```

当一切都不顺利时，有时甚至在那之前，我们只能选择使用SQL编写查询。

5.13. 原生SQL查询

HQL是一种强大的语言，它帮助减少了SQL的冗长，并且极大地增加了在不同数据库之间查询的可移植性。但最终，ORM的真正价值并不在于避免使用SQL，而是在于减轻我们在将SQL结果集带回Java程序后处理SQL数据所需的困扰。正如我们一开始所说的，Hibernate生成的SQL旨在与手写的SQL一起使用，而原生SQL查询就是我们提供的一个便利设施，使这一过程变得简单。

表格 38. 执行SQL

类型	Session方法	EntityManager方法	查询执行方法
选择查询	`createNativeQuery(String,Class)`	`createNativeQuery(String,Class)`	`getResultList()`, `getSingleResult()` 或 `getSingleResultOrNull()`
变更查询	`createNativeMutationQuery(String)`	`createNativeQuery(String)`	`executeUpdate()`
存储过程	`createStoredProcedureCall(String)`	`createStoredProcedureQuery(String)`	`execute()`

对于最简单的情况，Hibernate可以推断结果集的形状：

```
Book book = session.createNativeQuery("select * from Books where isbn = ?1", Book.class)
.getSingleResult();

String title = session.createNativeQuery("select title from Books where isbn = ?1", String.class)
.getSingleResult();
```

然而，通常情况下，JDBC的ResultSetMetaData中没有足够的信息来推断列与实体对象的映射关系。因此，在更复杂的情况下，您需要使用`@SqlResultSetMapping`注解定义一个命名映射，并将该名称传递给`createNativeQuery()`。这变得相当混乱，我们不想用一个例子来让您的眼睛受伤。

默认情况下，Hibernate在执行原生查询之前不会刷新会话（session）。这是因为会话不知道哪些内存中的修改会影响查询的结果。

所以，如果Books中有未刷新的更改，这个查询可能会返回陈旧的数据：

```
List<Book> books =  session.createNativeQuery("select * from Books")            .getResultList()
```

有两种方法可以确保在执行此查询之前刷新持久化上下文。

首先，我们可以通过调用`flush()`方法或将刷新模式设置为`ALWAYS`来强制刷新：

```
List<Book> books = session.createNativeQuery("select * from Books")
            .setHibernateFlushMode(ALWAYS)
            .getResultList()
```

或者，我们可以告诉Hibernate哪些修改状态会影响查询的结果：

```
List<Book> books =  session.createNativeQuery("select * from Books")
            .addSynchronizedEntityClass(Book.class)
            .getResultList()
```

> 您可以使用`createStoredProcedureQuery()`或`createStoredProcedureCall()`调用存储过程。

5.14. 查询结果的限制、分页和排序

如果一个查询可能返回的结果比我们一次处理的能力要多，我们可以指定：

1. 返回的最大行数的限制，和
2. 可选地，一个偏移量，有序结果集中要返回的第一行。

> 偏移量用于分页查询结果。

在HQL或原生SQL查询中，添加限制或偏移量有两种方式：

1. 使用查询语言本身的语法，例如，`offset 10 rows fetch next 20 rows only`，或
2. 使用SelectionQuery接口的`setFirstResult()`和`setMaxResults()`方法。

如果限制或偏移量是带参数的，第二种方法比较简单。例如，这个例子：

```
List<Book> books =  session.createSelectionQuery("from Book where title like ?1 order by title")
            .setParameter(1, titlePattern)
            .setMaxResults(MAX_RESULTS)
            .getResultList();
```

比这个例子更简单：

```
List<Book> books = session.createSelectionQuery("from Book where title like ?1 order by title fetch first ?2
rows only")
            .setParameter(1, titlePattern)
            .setParameter(2, MAX_RESULTS)
            .getResultList();
```

Hibernate的SelectionQuery在分页查询结果时有一种稍微不同的方式：

```
List<Book> books = session.createSelectionQuery("from Book where title like ?1 order by title")
            .setParameter(1, titlePattern)
            .setPage(Page.first(MAX_RESULTS))
            .getResultList();
```

一个密切相关的问题是排序。在分页查询时，通常需要按运行时确定的字段对查询结果进行排序。因此，SelectionQuery提供了一种选择，即指定查询结果应该按查询返回的实体类型的一个或多个字段排序：

```
List<Book> books = session.createSelectionQuery("from Book where title like ?1")
            .setParameter(1, titlePattern)
            .setOrder(List.of(Order.asc(Book._title), Order.asc(Book._isbn)))
            .setMaxResults(MAX_RESULTS)
            .getResultList();
```

不幸的是，使用JPA的TypedQuery接口没有办法实现这个功能。

表格 39. 查询限制、分页和排序的方法

方法名	作用	JPA标准
`setMaxResults()`	设置查询返回的结果数的限制	✓
`setFirstResult()`	设置查询返回结果的偏移量	✓
`setPage()`	通过指定Page对象设置限制和偏移量	✕
`setOrder()`	指定查询结果的排序方式	✕

5.15. 表达投影列表

投影列表是查询返回的结果项列表，即select子句中的表达式列表。由于Java没有元组类型，在Java中表示查询的投影列表一直是JPA和Hibernate的一个问题。传统上，我们大部分时候使用Object[]：

```
var results =
        session.createSelectionQuery("select isbn, title from Book", Object[].class)
            .getResultList();

for (var result : results) {
    var isbn = (String) result[0];
    var title = (String) result[1];
    ...
}
```

这实际上有点丑陋。现在，Java的record类型提供了一个有趣的替代方案：

```
record IsbnTitle(String isbn, String title) {}

var results =
        session.createSelectionQuery("select isbn, title from Book", IsbnTitle.class)
            .getResultList();

for (var result : results) {
    var isbn = result.isbn();
    var title = result.title();
    ...
}
```

请注意，我们能够在执行查询的那一行代码之前直接声明record类型。

现在，这只是在表面上更具类型安全性，因为查询本身在静态上没有被检查，所以我们不能说它在客观上更好。但也许你觉得它在美学上更令人愉悦。而且如果我们要在系统中传递查询结果，使用record类型要好得多。

与此问题相关的，Criteria查询API提供了一个更令人满意的解决方案。考虑以下代码：

```
var builder = sessionFactory.getCriteriaBuilder();
var query = builder.createTupleQuery();
var book = query.from(Book.class);
var bookTitle = book.get(Book_.title);
var bookIsbn = book.get(Book_.isbn);
var bookPrice = book.get(Book_.price);
query.select(builder.tuple(bookTitle, bookIsbn, bookPrice));
var resultList = session.createSelectionQuery(query).getResultList();
for (var result: resultList) {
    String title = result.get(bookTitle);
    String isbn = result.get(bookIsbn);
    BigDecimal price = result.get(bookPrice);
    ...
}
```

这段代码显然是完全类型安全的，远比我们在HQL中所能期望的要好。

5.16. 命名查询

@NamedQuery注解允许我们定义一个HQL查询，在启动过程中将其编译和检查。这意味着我们能够更早地发现查询中的错误，而不是等到查询实际执行时才发现。我们可以将@NamedQuery注解放在任何类上，甚至是实体类。

```
@NamedQuery(name="10BooksByTitle",
            query="from Book where title like :titlePattern order by title fetch first 10 rows only")
class BookQueries {}
```

我们必须确保带有@NamedQuery注解的类将被Hibernate扫描，方法有两种：

1. 在persistence.xml中添加 `<class>org.hibernate.example.BookQueries</class>`。
2. 调用 `configuration.addClass(BookQueries.class)`。

不幸的是，JPA的@NamedQuery注解无法放在包描述符上。因此，Hibernate提供了一个非常相似的注解 `@org.hibernate.annotations.NamedQuery`，可以在包级别指定。如果我们在包级别声明了命名查询，我们必须调用：

```
configuration.addPackage("org.hibernate.example")
```

这样Hibernate就知道在哪里找到它。

@NamedNativeQuery注解允许我们为本地SQL查询做同样的事情。使用@NamedNativeQuery的优势较少，因为Hibernate在本地数据库SQL方言中对查询的正确性几乎无法进行验证。

表40. 执行命名查询

类型	Session方法	EntityManager方法	查询执行方法
Selection	createNamedSelectionQuery(String,Class)	createNamedQuery(String,Class)	`getResultList()`, `getSingleResult()`, 或 `getSingleResultOrNull()`
Mutation	createNamedMutationQuery(String)	createNamedQuery(String)	executeUpdate()

我们执行命名查询的方式如下：

```
List<Book> books = entityManager.createNamedQuery(BookQueries_.QUERY_10_BOOKS_BY_TITLE)
            .setParameter("titlePattern", titlePattern)
            .getResultList();
```

这里，`BookQueries_.QUERY_10_BOOKS_BY_TITLE`是一个常量，其值为"10BooksByTitle"，由元模型生成器生成。

> 请注意，执行命名查询的代码并不知道查询是用HQL还是本地SQL编写的，这使得稍后更改和优化查询稍微容易一些。
>
> 在启动时检查我们的查询是很好的。但是，在编译时检查它们会更好。在"组织持久性逻辑"中，我们提到过，Metamodel Generator可以帮助我们实现编译时HQL查询字符串的验证，这也是使用@NamedQuery的原因之一。
>
> 但实际上，Hibernate还有一个单独的查询验证器，能够执行HQL查询字符串的编译时验证，这些字符串作为参数传递给createQuery()等方法。如果我们使用查询验证器，使用命名查询的优势就不那么明显了。

5.17. 控制按ID查找

我们可以通过HQL、Criteria或本地SQL查询几乎可以完成所有操作。但是，当我们已经知道所需实体的标识符时，使用查询可能会显得有些多余。而且，查询并不高效地利用了二级缓存。

我们之前介绍过`find()`方法。这是执行按ID查找的最基本方式。但正如我们之前看到的，它并不能完成所有操作。因此，Hibernate提供了一些API来简化特定的更复杂的查找操作：

表41. 按ID查找的操作

方法名	用途
`byId()`	允许我们通过EntityGraph指定关联抓取，正如我们所看到的；还可以指定一些额外选项，包括查找如何与二级缓存交互，以及实体是否应该以只读模式加载
`byMultipleIds()`	允许我们同时加载一批ID

当我们需要按ID检索同一实体类的多个实例时，批量加载非常有用：

```
var graph = session.createEntityGraph(Book.class);
graph.addSubgraph(Book_.publisher);

List<Book> books =
        session.byMultipleIds(Book.class)
            .withFetchGraph(graph)  // 控制关联抓取
            .withBatchSize(20)      // 指定显式批量大小
            .with(CacheMode.GET)    // 控制与缓存的交互
            .multiLoad(bookIds);
```

给定的`bookIds`列表将被分成批次，并且每个批次将在单个select语句中从数据库中获取。如果我们没有显式指定批量大小，将会自动选择一个批量大小。

我们还有一些用于处理自然ID查找的操作：

方法名	用途
`bySimpleNaturalId()`	适用于只有一个属性被标注为@NaturalId的实体
`byNaturalId()`	适用于多个属性被标注为@NaturalId的实体
`byMultipleNaturalId()`	允许我们同时加载一批自然ID

以下是如何通过复合自然ID检索实体的方法：

```
Book book = session.byNaturalId(Book.class)            .using(Book_.isbn, isbn)
.using(Book_.printing, printing)            .load();
```

请注意，这段代码片段完全是类型安全的，再次感谢Metamodel Generator。

5.18. 直接与JDBC交互

偶尔我们需要编写一些直接调用JDBC的代码。不幸的是，JPA没有很好的方法来做这个，但是Hibernate的Session可以实现这个需求。

```
session.doWork(connection -> {
    try (var callable = connection.prepareCall("{call myproc(?)}")) {
        callable.setLong(1, argument);
        callable.execute();
    }
});
```

传递给 `doWork()` 的Connection是Session正在使用的同一个连接，因此使用该连接执行的任何操作都在相同的事务上下文中。

如果需要返回值，可以使用 `doReturningWork()` 而不是 `doWork()`。

在一个由容器管理事务和数据库连接的环境中，获取JDBC连接可能不是最简单的方法。

5.19. 当事情出错时该怎么办

对象/关系映射被称为计算机科学的"越战"。做出这个类比的人是美国人，所以我们可以推测他可能是想暗示某种无法取胜的战争。这是相当具有讽刺意味的，因为在他发表这个评论的瞬间，Hibernate已经在取得胜利的边缘。

今天，越南是一个和平的国家，人均GDP在迅速增长，ORM问题也得到了解决。尽管如此，Hibernate是复杂的，ORM对于经验不足的人来说仍然存在许多陷阱，甚至有时也会困扰经验丰富的人。有时候，事情会出错。

在这一部分，我们将快速概述一些避免"泥潭"的一般策略。

1. 理解SQL和关系模型。了解你的关系型数据库的功能。如果你有幸有DBA（数据库管理员），请与他们紧密合作。Hibernate不是为Java对象提供"透明持久化"的解决方案。它是关于使两个出色的技术顺利协同工作。
2. 记录Hibernate执行的SQL语句。只有当你实际检查了正在执行的SQL语句后，你才能确定你的持久性逻辑是否正确。即使在一切似乎"正常"的时候，可能仍然存在潜在的N+1查询问题。
3. 修改双向关联时要小心。原则上，你应该更新关联的两端。但Hibernate并没有严格执行这一点，因为在某些情况下，这样的规则可能太过严格。不管怎样，你需要自己来维护模型的一致性。
4. 永远不要在不同线程或并发事务之间泄漏持久化上下文。制定策略或使用框架来确保这种情况永远不会发生。
5. 运行返回大结果集的查询时，要考虑会话缓存的大小。考虑使用无状态会话（stateless session）。
6. 深入思考二级缓存的语义，以及缓存策略对事务隔离性的影响。
7. 避免使用不需要的高级功能。Hibernate功能非常丰富，这是一件好事，因为它满足了大量用户的需求，其中许多用户只有一两个非常特殊的需求。但是没有人拥有所有那些特殊的需求。很可能，你一个也没有。以最合理的方式编写你的领域模型，使用最简单的映射策略。
8. 当某些事情的行为不符合你的期望时，简化。隔离问题。在在线寻求帮助之前，找到能够重现行为的最小测试用例。大多数时候，隔离问题本身就可能会提示一个明显的解决方案。
9. 避免使用"包装"JPA的框架和库。如果有一个对Hibernate和ORM有时确实成立的批评，那就是它使你距离JDBC的直接控制更远。额外增加的一层只会使你更加远离底层。
10. 避免从随机博主或Stack Overflow上复制/粘贴代码。你在网上找到的许多建议通常都不是最简单的解决方案，而且许多并不适用于Hibernate 6。相反，了解你在做什么；研究你正在使用的API的Javadoc；阅读JPA规范；遵循我们在本文档中给出的建议；直接与Zulip上的Hibernate团队联系（当然，我们有时可能有点暴躁，但我们始终希望你能够成功）。
11. 怀疑一切。你不必在所有情况下都使用Hibernate。