第 13 章

优化抓取和高速缓存

本章内容

- 全局抓取策略
- 高速缓存理论
- 高速缓存实践

本章将介绍如何从数据库中获取对象，以及当你在应用程序中逐个导航对象时如何优化对象网络的加载。

之后讲述高速缓存，介绍如何在本地应用程序和分布式应用程序中加快数据的获取。

13.1 定义全局抓取计划

从数据库中获取持久化对象是使用Hibernate最值得关注的部分之一。

13.1.1 对象获取选项

Hibernate提供了下列方法从数据库中获取对象：

- 导航对象图，从一个已经加载的对象开始，通过如`aUser.getAddress().getCity()`等这样的属性访问方法访问被关联的对象。如果持久化上下文仍然是打开的，当调用访问方法时，Hibernate就会自动加载（和预加载）对象图中的节点。
- 通过标识符获取，当一个对象的唯一标识符值已知时，这是最方便的方法。
- HQL（Hibernate Query Language，Hibernate查询语言），它是一种完全面向对象的查询语言。Java QL（Java Persistence query language，JPA持久化查询语言）是HQL的一个标准子集。
- Hibernate `Criteria`接口，它提供了一种类型安全和面向对象的方式来执行查询，而不需要进行字符串操作。这种机制包括基于示例对象的查询。
- 原生的SQL查询，包括存储过程的调用（在这里Hibernate仍然把JDBC的结果集映射到持久化对象图中）。

在Hibernate或者JPA应用程序中，可以结合使用这些方法。

本章并不详细讨论每一种获取方法。我们更感兴趣的是所谓的默认抓取计划（fetch plan）和抓取策略（fetching strategy）。默认的抓取计划和抓取策略是应用到特定的实体关联或者集合的计划和策略。换句话说，它定义了当加载自己的实体对象，以及访问一个被关联的对象或者集合时，被关联的对象或者集合是否以及如何被加载。每一种获取方法都可能使用不同的计划和策略——也就是说，计划定义了持久化对象网络的哪一部分应该被获取和怎样获取。你的目标是为应用程序中的每个用例找到最好的获取方法和抓取策略；同时也要最小化SQL查询的次数，以获得最好的性能。

在探讨抓取计划选项和抓取策略之前，我们将全面概括获取方法。（我们也会偶尔提到Hibernate高速缓存系统，本章稍后会全面探讨它。）

前一章已经介绍了如何通过标识符获取对象，这里不再赘述。我们直接讨论更灵活的查询选项，HQL（等同于JPA QL）和Criteria。这两者都允许创建任意查询。

1. HQL和JPA QL

HQL是常见的数据库查询语言SQL的一种面向对象的方言。HQL与ODMG OQL极其相似，但与OQL不同的是，它适用于SQL数据库，并且更易于学习（由于它很类似于SQL），并且实现更完整（没有任何OQL的实现是完整的）。

EJB 3.0标准定义了Java Persistence查询语言。这个新的JPA QL和HQL已经融合，并且JPA QL成了HQL的一个子集。有效的JPA QL查询也始终是有效的HQL查询。HQL有更多的选项，应该被认为是标准化子集的供应商扩展。

HQL常用于对象获取，而不是更新、插入或者删除数据。对象状态同步是持久化管理器的工作，而不是开发人员的工作。但是正如前一章中讲过的，如果用例（大量数据操作）需要，HQL和JPA QL都支持直接的大批量操作，用于更新、删除和插入。

大多数情况下，只需要获取特定类的对象，并通过该类的属性进行限制。例如，下列查询通过名字获取了用户。

```
Query q = session.createQuery(
    "from User as u where u.firstname = :fname"
);
q.setString("fname", "John");
List result = q.list();
```

在准备了查询q之后，把一个值绑定到具名参数fname。结果返回User对象的List。

HQL是强大的，即使可能一直不使用更高级的特性，但对于更难的问题也会需要它们。例如，HQL支持：

- 对按引用相关或者处于集合中的被关联对象的属性，应用限制的能力（用查询语言导航对象图）。
- 只获取一个或者多个实体的属性的能力，没有加载实体本身到持久化上下文的系统开销。有时这被称作报告查询，更准确的称呼是投影。
- 对查询结果进行排序的能力。
- 对查询结果进行分页的能力。

- 与group by、having和聚集函数（如sum、min和max/min）联合使用。
- 当每一行获取多个对象时使用外部联结。
- 调用标准的和用户定义的SQL函数的能力。
- 子查询（嵌套查询）。

第14章和第15章将讨论所有这些特性，以及可选的原生SQL查询机制。

2. 按条件查询

Hibernate的按条件查询（Query By Criteria，QBC）API允许查询在运行时通过Criteria对象的操作进行创建。这样让你可以动态地指定约束，而无需直接的字符串操作，但是会失去少量的HQL的灵活性或者功能性。另一方面，用条件表达的查询通常比用HQL表达的查询更不具备可读性。

用Criteria对象根据名字获取用户很容易，如：

```
Criteria criteria = session.createCriteria(User.class);
criteria.add( Restrictions.like("firstname", "John") );
List result = criteria.list();
```

一个Criteria就是一棵Criterion实例的树。Restriction类提供了返回Criterion实例的静态工厂方法。一旦建立了想要的Criteria树，就可以依靠数据库来执行它了。

许多开发人员更喜欢按条件查询，原因是他们认为它是更面向对象的方法。他们也喜欢其查询语法是在编译时被解析和验证的，相比之下，HQL表达式只有在运行时（或者启动时，如果使用了具体命名的查询）才可被解析。

Hibernate Criteria API好的一面是，Criterion框架允许用户进行扩展，这在像HQL的查询语言中很难做到。

注意，Criteria API是Hibernate固有的，它不是Java 持久化标准的一部分。在实践中，Criteria将成为JPA应用程序中最常用的Hibernate扩展。我们期待JPA或者EJB标准的未来版本中能包含类似的编程式的查询接口。

3. 按示例查询

作为Criteria工具的一部分，Hibernate支持按示例查询（Query By Example，QBE）。按示例查询背后的思想是，应用程序通过某些属性值设置，提供被查询类的实例（到非默认的值）。查询结果返回带有匹配属性值的所有持久化实例。QBE不是特别强大的方法。然而，它对于某些应用程序很方便，尤其当它结合Criteria使用的时候：

```
Criteria criteria = session.createCriteria(User.class);

User exampleUser = new User();
exampleUser.setFirstname("John");
criteria.add( Example.create(exampleUser) );

criteria.add( Restrictions.isNotNull("homeAddress.city") );

List result = criteria.list();
```

这个示例先创建查询User对象的新的Criteria。然后只使用firstname属性设置来添加Example对象User实例。最后，在执行这个查询之前添加Restriction条件。

QBY的一个典型用例是搜索屏，允许用户指定一系列不同的属性值要与返回的结果集相匹配。这种功能可能难以用查询语言表达清楚；需要字符串操作来指定约束的动态集合。

Criteria API和按示例查询机制会在第15章中进行更深入的讨论。

你现在知道Hibernate中基本的获取选项了。本节剩下的篇幅，关注对象抓取计划和抓取策略。

让我们从“什么应该被加载到内存中去”这个定义开始。

13.1.2 延迟的默认抓取计划

Hibernate给所有的实体和集合默认一个延迟的抓取策略。这意味着Hibernate在默认情况下只加载你正在查询的对象。我们用几个示例对此进行探讨。

如果查询Item对象（假设你通过它的标识符加载），则只有这个Item被加载到内存中去：

```
Item item = (Item) session.load(Item.class, new Long(123));
```

通过标识符的这个获取形成了单个（或者可能几个，如果继承或者二级表被映射的话）获取Item实例的SQL语句。在持久化上下文中，现在你在内存中便有了这个处于持久化状态的item对象，如图13-1所示。

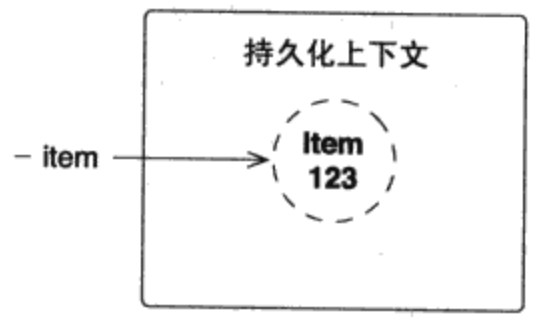

图13-1 一个未对Item实例初始化的占位符

我们骗了你。load()操作之后，内存中并没有持久化的item对象。甚至加载Item的SQL也没有被执行。Hibernate创建了一个看起来跟真的一样的代理（proxy）。

13.1.3 理解代理

代理是在运行时生成的占位符。每当Hibernate返回实体类的实例时，它就检查是否可以返回一个代理来代替，并避免数据库命中。代理是当真实对象第一次被访问时触发其加载的占位符：

```
Item item = (Item) session.load(Item.class, new Long(123));
item.getId();
item.getDescription(); // Initialize the proxy
```

这个示例中的第三行触发了把Item获取到内存的SQL的执行。只要你只访问数据库标识符属性，就没有必要初始化代理。（注意，如果用直接的字段访问映射标识符属性，就不是这样了；Hibernate当时甚至不知道getId()方法的存在。如果调用它，代理就必须被初始化。）

如果需要Item只是用来创建一个引用，代理就很有用了，例如：

```
Item item = (Item) session.load(Item.class, new Long(123));
User user = (User) session.load(User.class, new Long(1234));

Bid newBid = new Bid("99.99");
newBid.setItem(item);
newBid.setBidder(user);

session.save(newBid);
```

先加载两个对象：Item和User。Hibernate并没有命中数据库来完成这项工作，而是返回两个代理。这就是你所要的，因为你只需要Item和User来创建一个新的Bid。save(newBid)调用执行INSERT语句，通过Item和User的外键值保存BID表中的行——这是代理能够且必须提供的一

切。前一个代码片段没有执行任何SELECT语句！

如果调用get()而不是load()，就触发了一次数据库命中，且没有返回任何代理。get()操作始终命中数据库（如果实例还没有处在持久化上下文中，并且如果没有任何透明的二级高速缓存是活动的），如果无法找到对象就返回null。

JPA提供程序可以利用代理实现延迟加载。EntityManager API中相当于load()和get()这些操作的方法名称是find()和getReference()：

```
Item item = em.find(Item.class, new Long(123));

Item itemRef = em.getReference(Item.class, new Long(1234));
```

第一个调用find()，必须命中数据库以初始化Item实例。不允许代理——相当于Hibernate的get()操作。第二个调用getReference()，可能返回一个代理，但是不一定——它在Hibernate中变成load()。

由于Hibernate代理是实体类运行时生成的子类的实例，你无法通过一般的操作符得到一个对象的类。这就是辅助方法HibernateProxyHelper.getClassWithoutInitializingProxy(o)派上用场的地方。

假设你让Item实例进到内存中，要么通过显式获取，要么通过调用它的其中一个属性并强制代理初始化。持久化上下文现在包含一个完全加载的对象，如图13-2所示。

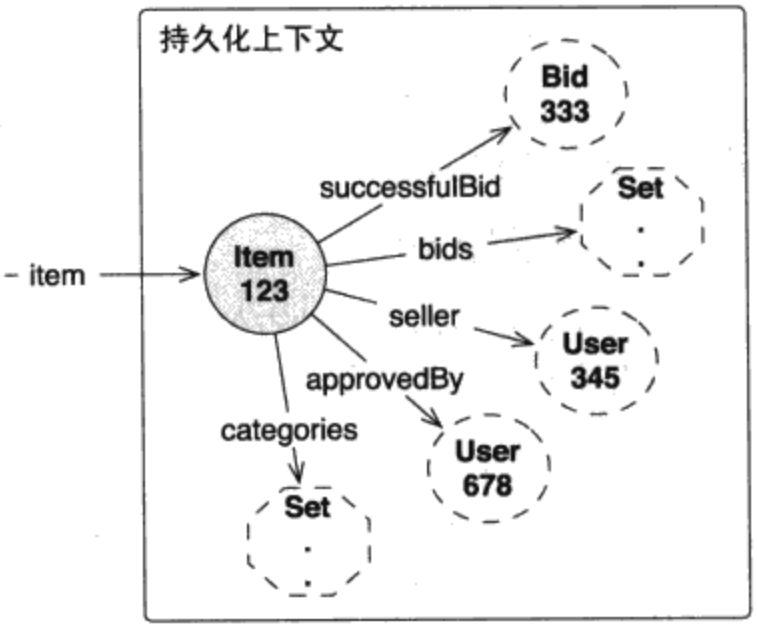

图13-2　代理和集合包装器表示被加载图的范围

在这幅图中你可以再次见到代理。这一次，这些是已经为所有*-to-one关联生成的代理。被关联的实体对象没有马上被加载；代理只提供标识符值。从一个不同的视角来看：标识符值是item行中所有的外键列。集合也没有被立即加载，但我们用术语集合包装器（collection wrapper）来描述这种占位符。Hibernate内部有一组智能的集合，可以按需要初始化它们自己。Hibernate用这些替换你的集合；这就是为什么应该只在领域模型中使用集合接口的原因。默认情况下，Hibernate给所有的关联和集合都创建占位符，并且仅仅立即获取值类型的属性和组件。（不幸的是，这并不是Java Persistence标准化的默认抓取计划；稍后会回来讨论其中的差别。）

常见问题　一对一关联的延迟加载有用吗？对一对一关联延迟加载，有时候把刚刚接触Hibernate的用户们搞糊涂了。如果你基于共享的主键来考虑一对一关联（见7.1.1节），关联只有当它为constrained="true"时才可以被代理。例如，Address始终有一个对User的引用。如果这个关联可为空并且可选，Hibernate首先必须命中数据库，决定应该应用代理还是null——延迟加载的目的是根本不命中数据库。可以通过BCI和拦截启用延迟加载，我们稍后会讨论到。

如果你调用任何不是标识符获取方法的方法，代理就被初始化，如果你通过集合的元素启动迭代，或者如果调用任何集合管理的操作，如size()和contains()，集合就被初始化。Hibernate提供一项对大集合最有用的额外设置：它们可以被映射为extra延迟。例如，考虑Item的bids集合：

```
<class name="Item" table="ITEM">
    ...
    <set name="bids"
         lazy="extra"
         inverse="true">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
</class>
```

集合包装器现在比以前更智能了。如果调用size()、contains()或者isEmpty()，集合将不再被初始化——查询数据库来获取必要的信息。如果它是一个Map或者List，containsKey()和get()操作也直接查询数据库。Hibernate扩展注解启用了同样的优化：

```
@OneToMany
@org.hibernate.annotations.LazyCollection(
    org.hibernate.annotations.LazyCollectionOption.EXTRA
)
private Set<Bid> bids = new HashSet<Bid>();
```

我们来定义不完全延迟的抓取计划。首先，可以给实体类禁用代理生成。

13.1.4 禁用代理生成

代理是个好东西：它们允许你只加载真正需要的数据。它们甚至让你创建对象之间的关联，而不用没有必要地命中数据库。有时候你会需要一种不同的计划——例如，你想要表达User对象应该永远被加载到内存中，并且不应该返回任何占位符来代替。

可以给一个特定的实体类禁用代理生成，通过XML映射元数据中的lazy="false"属性：

```
<class name="User"
       table="USERS"
       lazy="false">
       ...
</class>
```

JPA标准不要求通过代理实现；“代理”一词甚至没有出现在规范中。Hibernate是默认情况下依赖代理的JPA提供程序，因此禁用Hibernate代理的切换可以用一个供应商扩展：

```
@Entity
@Table(name = "USERS")
@org.hibernate.annotations.Proxy(lazy = false)
public class User { ... }
```

给实体禁用代理生成会有严重的后果。所有这些操作都需要数据库命中：

```
User user = (User) session.load(User.class, new Long(123));
User user = em.getReference(User.class, new Long(123));
```

User对象的load()无法返回代理。JPA操作getReference()可以不再返回代理引用。这可

能会是你想要实现的东西。然而，禁用代理对于所有引用该实体的关联来说，也有影响。例如，Item实体有一个对User的seller关联。考虑下列获取Item的操作：

```
Item item = (Item) session.get(Item.class, new Long(123));
Item item = em.find(Item.class, new Long(123));
```

除了获取Item实例之外，get()操作现在也加载Item的被链接的seller；没有给这个关联返回任何User代理。对于JPA也一样：已经用find()加载的Item没有引用seller代理。正在出售Item的User必须被立即加载。（我们稍后会解答这如何抓取的问题。）

在全局的级别上禁用代理生成经常过于粗粒度。通常，你只想禁用一个特定实体关联或者集合的延迟加载行为，来定义一个细粒度的抓取计划。你想要的正好相反：特定关联或者集合的即时加载。

13.1.5　关联和集合的即时加载

你已经知道，Hibernate默认情况下是延迟的。如果加载一个实体对象，所有被关联的实体和集合就都没有被初始化。一般来说，开发人员经常想要相反的结果：指定一个特定的实体关联或者集合应该始终被加载。你想要保证这个数据在内存中不用额外的数据库命中就可用。最重要的是，比如，即使Item实例处于脱管状态，你希望仍然可以访问Item的seller。必须定义这项抓取计划，这是你想要始终加载到内存中的一部分对象网络。

假设你始终需要Item的seller。在Hibernate XML映射元数据中，映射从Item到User的关联为lazy="false"：

```
<class name="Item" table="ITEM">
    ...
    <many-to-one name="seller"
                 class="User"
                 column="SELLER_ID"
                 update="false"
                 not-null="true"
                 lazy="false"/>
    ...
</class>
```

同样的“始终加载”保证可以被应用到集合——例如，Item的所有bids：

```
<class name="Item" table="ITEM">
    ...
    <many-to-one name="seller" lazy="false" .../>
    <set name="bids"
         lazy="false"
         inverse="true">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
    ...
</class>
```

如果现在get()一个Item（或者强制初始化一个被代理的Item），seller对象和所有的bids都作为持久化实例被加载到你的持久化上下文中去：

```
Item item = (Item) session.get(Item.class, new Long(123));
```

这个调用之后的持久化上下文如图13-3所示。

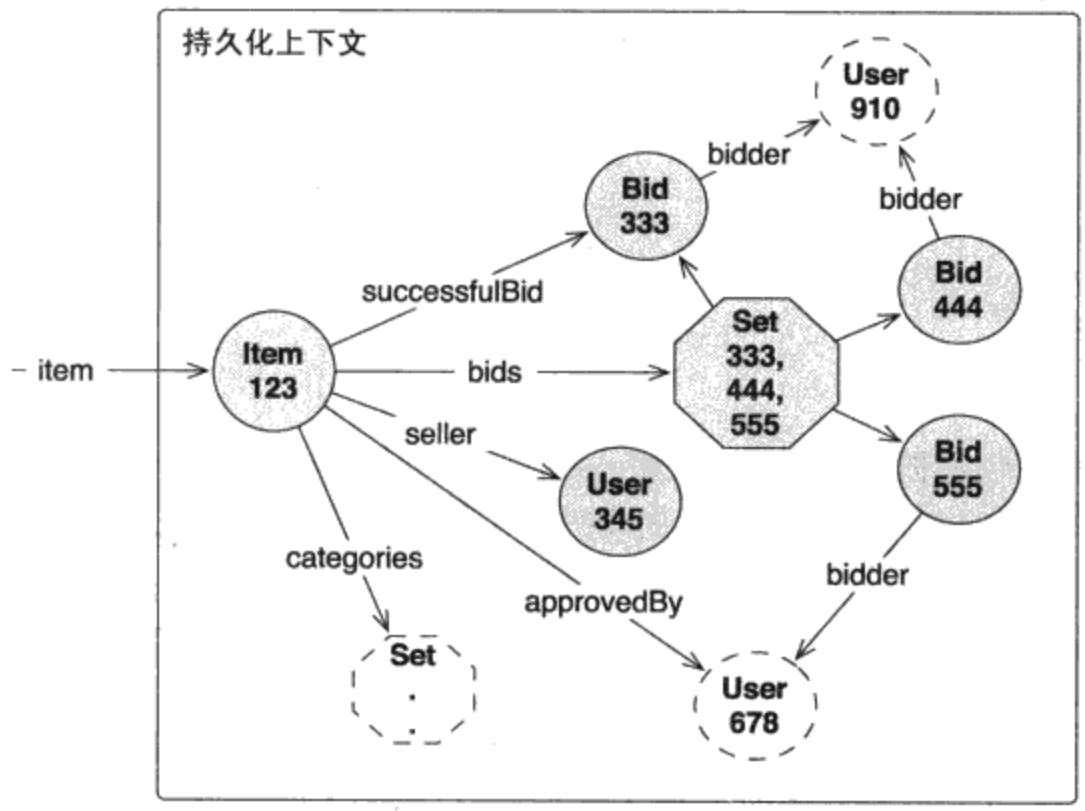

图13-3 通过禁用延迟关联和集合，即时抓取的一幅更大的图

其他延迟的被映射关联和集合（如每个Bid实例的bidder）还是没有被初始化，并且你一访问它们就立即被加载。想象你在加载Item之后关闭持久化上下文。现在可以在脱管状态下导航到Item的seller，并给这个Item遍历所有的bids。如果导航到这个Item要被分配到的categories，你就得到了LazyInitializationException！很显然，这个集合并非你抓取计划的一部分，并且没有在持久化上下文关闭之前初始化。如果试图访问一个代理（例如，核准项目的User），也会发生这种情况。（注意，访问这个代理可以用两种方法：通过approveBy和bidder引用。）

利用注解，转换一个实体关联或者集合的FetchType来得到同样的结果：

```
@Entity
public class Item {

    ...

    @ManyToOne(fetch = FetchType.EAGER)
    private User seller;

    @OneToMany(fetch = FetchType.EAGER)
    private Set<Bid> bids = new HashSet<Bid>();

    ...
}
```

在Hibernate中，FetchType.EAGER提供与lazy="false"同样的保证：被关联的实体实例必须被即时抓取，而不是延迟。我们已经注意到，Java Persistence有一项与Hibernate不同的默认抓取计划。虽然Hibernate中的所有关联都是完全延迟的，所有@ManyToOne和@OneToOne关联都默认为FetchType.EAGER！这项默认被标准化，允许Java Persistence提供程序不通过延迟加载实现（在实践中，这样的持久化提供程序并没有太大的用处）。建议你通过在“对一”的关联映射中设置FetchType.LAZY，默认为Hibernate的延迟加载计划，并且只在必要时才覆盖它：

```
@Entity
public class Item {

    ...

    @ManyToOne(fetch = FetchType.LAZY)
    private User seller;

    ...
}
```

你现在知道如何创建抓取计划了；也就是说，如何定义持久化对象网络的哪一部分应该被获取到内存中。在介绍这些对象应该如何被加载的定义方式，以及如何优化将要被执行的SQL之前，我们想要示范另一种不依赖代理的延迟加载策略。

13.1.6 通过拦截延迟加载

Hibernate提供的运行时代理生成，是透明的延迟加载的一种极好的选择。这个实现公开的唯一必要条件是一个包，或者是必须被代理的类中公有可见的无参构造器，以及非final方法和类声明。在运行时，Hibernate生成一个充当代理类的子类；这用私有的构造器或者final实体类是不可能的。

另一方面，许多其他的持久化工具没有使用运行时代理：它们使用拦截。对于你为什么要在Hibernate中使用拦截而不是运行时代理生成，没有很好的解释。非私有的构造器条件的确没什么大不了。但是有两种情况，你可能不想使用代理：

- 运行时代理不完全透明的唯一情况是通过instanceof测试的多态关联。或者，你可能想要依据类型转换一个对象，但是无法进行，因为代理是运行时生成的子类的一个实例。7.3.1节已介绍过如何避免这个问题，以及如何处理这个问题。用拦截代替代理也可以解决这些问题。
- 代理和集合包装器只能被用于延迟加载实体关联和集合。它们无法用来延迟加载单独的标量属性或者组件。我们认为这种优化将没有什么用。例如，你通常不想要延迟加载Item的initialPrice。如果没有使用（a）非常大量的可选列，或者没有使用（b）包含了必须按需获取的大值的可选列，就没有必要在SQL中被选中的单独列的级别上进行优化。大值最好用定位器对象表示；按照定义，它们提供延迟加载，无需拦截。然而，拦截（通常还有代理）还可以帮助优化列读取。

让我们通过几个示例来讨论对延迟加载的拦截。

想象你不想利用User实体类的代理，但仍然想要把一个关联延迟加载到User（例如，Item

的seller）的这一好处。用no-proxy映射这个关联：

```
<class name="Item" table="ITEM">
    ...
    <many-to-one name="seller"
                 class="User"
                 column="SELLER_ID"
                 update="false"
                 not-null="true"
                 lazy="no-proxy"/>
    ...
</class>
```

lazy属性默认为proxy。通过设置no-proxy，你在告诉Hibernate对这个关联应用拦截：

```
Item item = (Item) session.get(Item.class, new Long(123));
User seller = item.getSeller();
```

第一行把一个Item对象获取到持久化状态中去。第二行访问Item的seller。对getSeller()的这个调用被Hibernate拦截，并触发正在讨论的该User的加载。注意代理如何比拦截更延迟：可以调用item.getSeller().getId()而不强制代理的初始化。如果你只想要设置引用，这就使得拦截没有多大用处了，就像我们前面讨论过的。

也可以延迟加载通过<property>或者<component>映射的属性；在Hibernate XML映射中，这里启用拦截的属性是lazy="true"。利用注解，@Basic(fetch = FetchType.LAZY)是给Hibernate的一个提示：有一个属性或者组件应该通过拦截被延迟加载。

为了利用注解给关联禁用代理并启用拦截，必须依赖Hibernate扩展：

```
@ManyToOne
@JoinColumn(name="SELLER_ID", nullable = false, updatable = false)
@org.hibernate.annotations.LazyToOne(
    org.hibernate.annotations.LazyToOneOption.NO_PROXY
)
private User seller;
```

为了启用拦截，类的字节码必须在编译之后、运行时之前被使用。Hibernate提供一个Ant任务用于这一目的：

```
<target name="instrument" depends="compile">

    <taskdef name="instrument"
             classname=
➥ "org.hibernate.tool.instrument.cglib.InstrumentTask"
             classpathref="project.classpath"/>

    <instrument verbose="true">
        <fileset dir="${build.dir}/my/entity/package/">
            <include name="*.class"/>
        </fileset>
    </instrument>

</target>
```

你自己要决定是否想给延迟加载使用拦截——依据我们的经验，好的用例很少。

一般来说，你不仅应当定义持久化对象网络的什么部分必须被加载，还应当定义这些对象如何获取。除了创建抓取计划之外，你还应当通过正确的抓取策略对它进行优化。

13.2　选择抓取策略

Hibernate执行SQL SELECT语句把对象加载到内存。如果加载一个对象，那么就要执行单个或者几个SELECT，这取决于所涉及的表的数量，以及你所采用的抓取策略。

你的目标是把SQL语句的数量减到最少，简化SQL语句，以便尽可能地提高查询效率。你通过给每个集合或者关联应用最佳的抓取策略来做到这一点。让我们一步步地探讨不同的选项。

默认情况下，每当访问被关联的对象和集合时（如果你使用Java Persistence，假设你映射所有的“对一”关联为FetchType.LAZY），Hibernate就延迟抓取它们。看看下列这个繁琐的代码示例：

```
Item item = (Item) session.get(Item.class, new Long(123));
```

你没有配置任何关联或者集合为非延迟，也没有配置可以为所有关联生成的代理。因而，这项操作生成了下列SQL SELECT语句：

```
select item.* from ITEM item where item.ITEM_ID = ?
```

（注意，Hibernate生成的真正SQL自动包含生成的别名；为了增加可读性，我们已经在接下来的所有示例中把它们移除了。）你会看到SELECT只查询ITEM表，并获取一个特定的行。所有实体关联和集合都没有被获取。如果你访问任何被代理的关联或者未被初始化的集合，就会执行第二个SELECT以便按需获取数据。

第一个优化步骤是，减少利用默认的延迟行为时定会看到的额外按需SELECT的数量——例如，通过预抓取数据。

13.2.1　批量预抓取数据

如果每一个实体关联和集合都仅仅按需抓取，完成一个特定的过程可能就需要许多额外的SQL SELECT语句了。例如，考虑下列获取所有Item对象的查询，并访问每件货品seller的数据：

```
List allItems = session.createQuery("from Item").list();

processSeller( (Item)allItems.get(0) );
processSeller( (Item)allItems.get(1) );
processSeller( (Item)allItems.get(2) );
```

一般来说，你在这里用了一个循环，并遍历结果，但这段代码所公开的问题是一样的。你看到1个SQL SELECT获取所有的Item对象，并且你一处理Item的每一个seller时立即就有另一个SELECT。所有被关联的User对象都是代理。这是我们将要深入阐述的最糟糕的案例场景之一：n+1查询问题。SQL看起来像这样：

```
select items...

select u.* from USERS u where u.USER_ID = ?
select u.* from USERS u where u.USER_ID = ?
```

```
select u.* from USERS u where u.USER_ID = ?
...
```

Hibernate提供了一些算法，可以预抓取User对象。我们现在讨论的第一个优化称作批量抓取（batch fetching），它像这样进行：如果User的1个代理必须被初始化，就在相同的SELECT中初始化几个代理。换句话说，如果你已经知道持久化上下文中有3个Item实例，并且它们已经把一个代理应用到了它们的seller关联，那么你可以也初始化所有代理而不只是1个。

批量抓取经常被称作瞎猜优化（blind-guess optimization），因为你不知道在一个特定的持久化上下文中，有多少个未被初始化的User代理。在前一个例子中，这个数量取决于返回的Item对象的数量。你猜测并把批量大小（batch-size）抓取策略应用到User类映射：

```
<class name="User"
       table="USERS"
       batch-size="10">
...
</class>
```

你在告诉Hibernate：如果必须初始化1个代理，就在单个SQL SELECT中最多预抓取10个未被初始化的代理。之前的查询和过程所产生的SQL现在看起来可能像这样：

```
select items...

select u.* from USERS u where u.USER_ID in (?, ?, ?)
```

获取所有Item对象的第一个语句在你list()查询的时候执行。下一个语句（获取3个User对象）在你一初始化由allItems.get(0).getSeller()返回的第一个代理时，就会被触发。这个查询同时加载3个卖主——因为这就是初始查询要返回的项目数量，以及当前的持久化上下文中未被初始化的代理数量。你定义了批量大小为“最多10个”。如果返回的货品超过10个，你就会看到第二个查询如何在一批中获取10个卖主。如果应用程序命中另一个尚未被初始化的代理，就会获取到另一批的10个，等等，直到持久化上下文中再也没有未被初始化的代理，否则应用程序停止访问被代理的对象。

常见问题 什么是真正的批量抓取算法？可以回想一下前面阐述过的批量抓取，但是如果你在实践中体验过，就会发现一种稍微不同的算法。是否想了解和理解这种算法，或者你是否信任Hibernate会正确地完成，这都取决于你。举个例子，想象一个批量大小为20，未被初始化的代理总数量为119，那么它们必须以多批加载。启动时，Hibernate读取映射元数据，并在内部创建11个批量加载器。每个加载器都知道它可以初始化多少个代理：20，10，9，8，7，6，5，4，3，2，1。目标是把加载器创建的内存消耗减到最少，并创建足够的可以生成每一个可能的批量抓取的加载器。另一个目标显然是把SQL SELECT的数量减到最少。为了初始化119个代理，Hibernate执行了7批（你或许以为是6批，因为6 × 20>119）。应用程序的批量加载器有五次为20个，一次为10个，一次为9个，由Hibernate自动选择。

批量抓取也可用于集合：

```
<class name="Item" table="ITEM">
```

```
    ...
    <set name="bids"
         inverse="true"
         batch-size="10">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

如果现在强制初始化1个bids集合，则立即加载相同类型的集合最多10个（如果它们在当前的持久化上下文中没有被初始化）：

```
select items...

select b.* from BID b where b.ITEM_ID in (?, ?, ?)
```

在这个例子中，你再一次拥有了3个处于持久化状态的Item对象，并且接触到了其中一个未被加载的bids集合。现在所有3个Item对象都让它们的bids在单个SELECT中加载。

对实体代理和集合的批量大小设置也可以通过注解使用，但是只能作为Hibernate扩展：

```
@Entity
@Table(name = "USERS")
@org.hibernate.annotations.BatchSize(size = 10)
public class User { ... }

@Entity
public class Item {
    ...
    @OneToMany
    @org.hibernate.annotations.BatchSize(size = 10)
    private Set<Bid> bids = new HashSet<Bid>();

    ...
}
```

通过批量策略预抓取代理和集合真的是一种瞎猜。正是这种聪明的优化可以显著地减少（对于初始化所使用的所有对象是必需的）SQL语句的数量。当然，预抓取的唯一缺点是，可能预抓取了最终不需要的数据。用更少的SQL语句的代价可能是更高的内存消耗。后者经常更为重要：内存很便宜，但是可伸缩的数据库服务器则不便宜。

另一种并非瞎猜的预抓取算法是使用子查询，通过单个语句初始化多个集合。

13.2.2　通过子查询预抓取集合

最后举一个示例，并应用一种（或许）更好的预抓取优化：

```
List allItems = session.createQuery("from Item").list();

processBids( (Item)allItems.get(0) );
processBids( (Item)allItems.get(1) );
processBids( (Item)allItems.get(2) );
```

你用一个初始的SQL SELECT获取所有的Item对象，并在每个bids集合被访问的时候用一个额外的SELECT。改善这种方法的一种可能是通过批量抓取；然而，需要算出试用的最佳批量大小。

一种更好的优化是这个集合映射的子查询抓取：

```
<class name="Item" table="ITEM">
    ...
    <set name="bids"
         inverse="true"
         fetch="subselect">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

一旦你强制初始化1个bids集合，Hibernate现在就立即给所有被加载的Item对象初始化所有bids集合。它通过在一个子查询中重新运行（被稍微修改的）第一个初始查询来完成：

```
select i.* from ITEM i

select b.* from BID b
    where b.ITEM_ID in (select i.ITEM_ID from ITEM i)
```

在注解中，必须再次使用一个Hibernate扩展启用这个优化：

```
@OneToMany
@org.hibernate.annotations.Fetch(
    org.hibernate.annotations.FetchMode.SUBSELECT
)
private Set<Bid> bids = new HashSet<Bid>();}
```

利用子查询预抓取是一种强大的优化；稍后当我们探讨一种典型的场景时，将介绍更详细的内容。在编写本书之时，子查询抓取只可用于集合，而不可用于实体代理。还要注意，作为子查询重新运行的原始查询，只被Hibernate为一个特定Session而记住。如果你分离一个Item实例，而没有初始化bids的集合，那么重附它并启动对该集合的遍历，则不会发生其他集合的预抓取。

如果你努力减少额外SELECT（对于利用延迟加载并按需获取对象和集合，它是自然要用的）的数量，那么所有前面的抓取策略都有帮助。最后一种抓取策略与按需获取相反。你经常想要在同一个初始的SELECT中，通过一个JOIN获取被关联的对象或者集合。

13.2.3 通过联结即时抓取

延迟加载是一种极好的默认策略。另一方面，你可以经常看看领域模型和数据模型，并说“每次我需要Item的时候，也需要该Item的seller”。如果可以把它变成语句，就应该到映射元数据中去，给seller关联启用即时抓取（eager fetching），并利用SQL联结：

```
<class name="Item" table="ITEM">
    ...
    <many-to-one name="seller"
                 class="User"
                 column="SELLER_ID"
                 update="false"
                 fetch="join"/>

</class>
```

现在Hibernate在单个SQL语句中把Item和它的seller都加载了。例如：

```
Item item = (Item) session.get(Item.class, new Long(123));
```

这个操作触发了下列SQL SELECT：

```
select i.*, u.*
from ITEM i
    left outer join USERS u on i.SELLER_ID = u.USER_ID
where i.ITEM_ID = ?
```

很显然，seller不再被按需延迟加载，而是立即加载。因此，fetch="join"禁用了延迟加载。如果你只用lazy="join"启用即时抓取，会立即看到第二个SELECT。通过fetch="join"，在同一个SELECT中加载seller。这个查询的结果集如图13-4所示。

select i.*, u.* from ITEM i
left outer join USERS u on i.SELLER_ID = u.USER_ID
where i.ITEM_ID = ?

ITEM_ID	DESCRIPTION	SELLER_ID	...	USER_ID	USERNAME	...
1	Item Nr. One	2	...	2	johndoe	...

图13-4　联结两张表来即时抓取被关联的行

Hibernate读取这个行，并封送来自结果的两个对象。它利用从Item到User的一个引用即seller关联，把它们连接起来。如果Item没有卖主，所有u.*列都会被填上NULL。这就是为什么Hibernate使用外部联结的原因，因此它不仅可以获取有卖主的Item对象，而且可以获取全部Item对象。但是你知道，在CaveatEmptor中Item必须有一个seller。如果你启用<many-to-one not-null="true"/>，Hibernate就会执行一个内部联结而不是外部联结。

也可以在集合上设置即时联结抓取策略：

```
<class name="Item" table="ITEM">
    ...
    <set name="bids"
         inverse="true"
         fetch="join">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```

如果现在加载许多个Item对象，例如通过creatCriteria(Item.class).list()，产生的SQL语句看起来像这样：

```
select i.*, b.*
from ITEM i
    left outer join BID b on i.ITEM_ID = b.ITEM_ID
```

现在这个结果集包含许多行，每个有许多个bids的Item都有重复数据，并且NULL过滤掉所有没有bids的Item对象。看看图13-5中的结果集。

select i.*, b.* from ITEM i
left outer join BID b on i.ITEM_ID = b.ITEM_ID

ITEM_ID	DESCRIPTION	...	BID_ID	ITEM_ID	AMOUNT
1	Item Nr. One	...	1	1	99.00
1	Item Nr. One	...	2	1	100.00
1	Item Nr. One	...	3	1	101.00
2	Item Nr. Two	...	4	2	4.99
3	Item Nr. Three	...	NULL	NULL	NULL

图13-5 被关联的集合元素的外部联结抓取

Hibernate创建了3个持久化的Item实例，以及4个Bid实例，并在持久化上下文中把它们全部链接在一起，以便你可以导航这个对象图，并遍历集合——甚至在持久化上下文被关闭，且所有对象被脱管的时候。

使用内部联结即时抓取集合在概念上是可能的，并且我们稍后也将在HQL查询中完成。然而，在映射元数据中去掉所有在全局抓取策略中没有bids的Item对象并没有意义，因此没有选项用于集合的全局内部联结即时抓取。

利用Java Persistence注解，通过FetchType注解属性启用即时抓取：

```
@Entity
public class Item {
    ...
    @ManyToOne(fetch = FetchType.EAGER)
    private User seller;

    @OneToMany(fetch = FetchType.EAGER)
    private Set<Bid> bids = new HashSet<Bid>();

    ...
}
```

这个映射示例看起来应该很熟悉：你前面用它来禁用关联和集合的延迟加载。Hibernate把它默认当作一个不应该通过立即的第二个SELECT（而是通过初始查询中的一个JOIN）执行的即时抓取。

可以保留FetchType.EAGER这个Java Persistence注解，但是通过显式添加一个Hibernate扩展注解，从联结抓取转换到一个立即的第二个查询：

```
@Entity
public class Item {

    ...

    @ManyToOne(fetch = FetchType.EAGER)
    @org.hibernate.annotations.Fetch(
        org.hibernate.annotations.FetchMode.SELECT
    )
    private User seller;

}
```

如果一个Item实例被加载，Hibernate将通过一个立即的第二个SELECT，即时加载这件货品的卖主。

最后，必须介绍一个全局的、可以用来控制被联结的实体关联（不是集合）最大数量的Hibernate配置设置。考虑你已经在映射元数据中设置所有多对一和一对多关联映射为fetch="join"（或者FetchType.EAGER）。假设Item有一个successfulBid关联，Bid有一个bidder，并且这个User有一个shippingAddress。如果所有这些关联都通过fetch="join"映射，那么当你加载Item时，要联结多少张表，要获取多少数据？

在这种情况下，被联结的表的数量取决于全局的hibernate.max_fetch_depth配置属性。默认情况下，没有设置限制，因此加载Item也在单个查询中获取Bid、User和Address。合理的设置很小，通常在1～5之间。你甚至可以通过设置该属性为0，给多对一和一对一的关联禁用联结抓取！（注意有些数据库方言可以预设这项属性，如MySQLDialect把它设置为2。）

如果涉及继承或者被联结的映射，SQL查询也会变得更加复杂。每当给一个特定的实体类映射二级表时，你需要考虑几个额外的优化选项。

13.2.4　给二级表优化抓取

如果你查询的对象属于继承层次结构一部分的类，SQL语句就变得更加复杂了：

```
List result = session.createQuery("from BillingDetails").list();
```

这项操作获取所有的BillingDetails实例。SQL SELECT现在取决于你已经给BillingDetails和它的子类CreditCard和BankAccount选择的继承映射策略。假设你已经把它们全部映射到一张表（每个层次结构一张表，table-per-hierarchy），这个查询与前一节介绍过的没有什么区别。然而，如果你已经利用隐式多态映射了它们，这个单个的SQL操作就可能根据每个子类的每张表产生几个SQL SELECT。

1. “每个子类一张表”层次结构的外部联结

如果以标准的方式映射层次结构（请见5.1.4节中的表和映射），那么在这个初始语句中，所有子类表都被外部联结：

```
select
    b1.BILLING_DETAILS_ID,
    b1.OWNER,
    b1.USER_ID,
    b2.NUMBER,
    b2.EXP_MONTH,
    b2.EXP_YEAR,
    b3.ACCOUNT,
    b3.BANKNAME,
    b3.SWIFT,
    case
        when b2.CREDIT_CARD_ID is not null then 1
        when b3.BANK_ACCOUNT_ID is not null then 2
        when b1.BILLING_DETAILS_ID is not null then 0
    end as clazz
from
```

```
        BILLING_DETAILS b1
    left outer join
        CREDIT_CARD b2
            on b1.BILLING_DETAILS_ID = b2.CREDIT_CARD_ID
    left outer join
        BANK_ACCOUNT b3
            on b1.BILLING_DETAILS_ID = b3.BANK_ACCOUNT_ID
```

这已经是个值得关注的查询了。它联结3张表，并利用一个CASE ...WHEN ... END表达式，用0~2之间的一个数填到clazz列中。然后Hibernate可以读取结果集，并在这个数字的基础上，决定被返回的每一行表示什么类的一个实例。

许多数据库管理系统限制可以与OUT JOIN合并的表的最大数目。如果你用标准的策略映射了一个很宽很深的继承层次结构，就可能碰到这一限制（我们正在讨论应该被重新考虑的继承层次结构以适应你毕竟正在使用的是SQL数据库的事实）。

2. 转换到额外的选择

然后在映射元数据中，你可以告诉Hibernate转换到一种不同的抓取策略。你想要继承层次结构的一些部分通过立即的额外SELECT语句而被抓取，而不是通过初始查询中的OUT JOIN。

启用这种抓取策略的唯一方式是稍微重构映射，作为使用了辨别标志列的每个层次结构一张表和使用了<join>映射每个子类一张表的混合：

```
<class name="BillingDetails"
       table="BILLING_DETAILS"
       abstract="true">

    <id name="id"
        column="BILLING_DETAILS_ID"
        .../>

    <discriminator
        column="BILLING_DETAILS_TYPE"
          type="string"/>

    ...
    <subclass name="CreditCard" discriminator-value="CC">
        <join table="CREDIT_CARD" fetch="select">
            <key column="CREDIT_CARD_ID"/>
            ...
        </join>
    </subclass>

    <subclass name="BankAccount" discriminator-value="BA">
        <join table="BANK_ACCOUNT" fetch="join">
            <key column="BANK_ACCOUNT_ID"/>
            ...

            </join>
    </subclass>

</class>
```

这个映射把CreditCard和BankAccount类分开到它们各自的表中，但是在超类表中保存辨别标志列。CreditCard对象的抓取策略是select，而BankAccount的策略则是默认的join。现

在，如果查询所有的BillingDetails，就会生成下列SQL：

```
select
    b1.BILLING_DETAILS_ID,
    b1.OWNER,
    b1.USER_ID,
    b2.ACCOUNT,
    b2.BANKNAME,
    b2.SWIFT,
    b1.BILLING_DETAILS_TYPE as clazz
from
    BILLING_DETAILS b1
    left outer join
        BANK_ACCOUNT b2
            on b1.BILLING_DETAILS_ID = b2.BANK_ACCOUNT_ID

select cc.NUMBER, cc.EXP_MONTH, cc.EXP_YEAR
from CREDIT_CARD cc where cc.CREDIT_CARD_ID = ?

select cc.NUMBER, cc.EXP_MONTH, cc.EXP_YEAR
from CREDIT_CARD cc where cc.CREDIT_CARD_ID = ?
```

第一个SQL SELECT从超类表和BANK_ACCOUNT表中获取所有的行。它也给每一行返回辨别标志值作为clazz列。Hibernate现在根据CREDIT_CARD表给第一个结果（这个结果有着CreditCard的正确辨别标志）的每一行执行一个额外的select。换句话说，两个查询意味着BILLING_DETAILS超类表中的两个行表示一个CreditCard对象（的一部分）。

这种优化几乎没有必要，但你现在也知道，每当处理<join>映射时，可以从一个默认的join抓取策略转换到一个额外的立即的select。

现在，我们已经通过你可以在映射元数据中设置的所有选项，来影响默认抓取计划和抓取策略的探讨。你学习了如何定义什么应该通过操作lazy属性进行加载，以及它应该如何通过设置fetch属性加载。在注解中，你使用FetchType.LAZY和FetchType.EAGER，并使用Hibernate扩展实现对抓取计划和策略更细粒度的控制。

知道了所有可用的选项只是实现优化的和有效的Hibernate或者Java Persistence应用程序的一小步。你还需要知道应用一种特定的策略的时机。

13.2.5　优化指导方针

默认情况下，Hibernate从不加载你没有请求的数据，这样减少了持久化上下文的内存消耗。然而，它也会让你面临所谓的n+1查询问题。如果每一个关联和集合都只按需初始化，并且没有配置其他的策略，特定的过程也可以很好地执行几十甚至几百个查询，以获得你需要的所有数据。你需要正确的策略来避免执行过多的SQL语句。

如果从默认的策略转换到通过联结即时抓取数据的查询，就可能遇到另一个问题：笛卡儿积问题。无须执行过多的SQL语句，相反现在可以（经常作为一种附带作用）创建获取过多数据的语句。

要找到这两个极端之间的中间地带：每个过程的正确抓取策略，以及你应用程序中的用例。你需要知道应该在映射元数据中设置哪种全局的抓取计划和策略，以及只对一个特定的查询应用

哪种抓取策略（通过HQL还是Criteria）。

现在介绍有关过多查询和笛卡儿积的几个基本问题，然后引导你一步步地完成优化。

1. n+1查询问题

n+1查询问题很容易通过一些示例代码理解。假设你没有在映射元数据中配置任何抓取计划或者抓取策略：所有的东西都是延迟的，且按需加载。下列示例代码试图给所有Item找到最高的Bid（当然，有许多其他的方法可以更轻松地实现这一点）：

```
List<Item> allItems = session.createQuery("from Item").list();
// List<Item> allItems = session.createCriteria(Item.class).list();

Map<Item, Bid> highestBids = new HashMap<Item, Bid>();

for (Item item : allItems) {
    Bid highestBid = null;
    for (Bid bid : item.getBids() ) { // Initialize the collection
        if (highestBid == null)
            highestBid = bid;
        if (bid.getAmount() > highestBid.getAmount())
            highestBid = bid;
    }
    highestBids.put(item, highestBid);
}
```

首先，获取所有Item实例；这在HQL和Criteria查询之间没有区别。这个查询触发1个SQL SELECT，获取ITEM表的所有行，并返回n个持久化对象。接下来，你遍历这个结果，并访问每一个Item对象。

你访问的是每个Item的bids集合。这个集合目前为止还没有被初始化，每件货品的Bid对象都必须通过一个额外的查询来加载。这整个代码片段因此产生了n+1查询。

你永远都想要避免n+1查询。

第一种解决方案可能是给集合改变你的全局映射元数据，启用批量预抓取：

```
<set name="bids"
     inverse="true"
     batch-size="10">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

不是n+1查询，你现在看到n/10+1查询在把必要的集合获取到内存中。这项优化对于一个拍卖应用程序来说似乎很合理："只在需要一件货品的出价时才按需加载它们。但是如果出价的1个集合必须针对某件特定的货品被加载，那么就假设持久化上下文中的其他货品对象也需要初始化它们的出价集合。批量完成这项工作，因为似乎并非所有的货品对象都需要它们的出价。"

利用一个基于子查询的预抓取，可以把选择的数量减到正好两个：

```
<set name="bids"
     inverse="true"
     fetch="subselect">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

过程中的第一个查询现在执行单个SQL SELECT，获取所有Item实例。Hibernate记住这个语句，并当你命中第一个未被初始化的集合时，再次应用它。所有的集合都通过第二个查询被初始化。这项优化的原因略有不同："只在需要一件货品出价的时候才按需加载它们。但是如果出价的1个集合必须针对某件特定货品加载，那么就假设持久化上下文中的所有其他货品对象也需要初始化它们的出价集合。"

最后，可以有效地关闭bids集合的延迟加载，并转换到只导致单个SQL SELECT的一种即时抓取策略：

```
<set name="bids"
     inverse="true"
     fetch="join">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

这似乎是一项你不应该进行的优化。你真的能说"每当需要一件货品时，也需要它的所有出价"吗？映射元数据中的抓取策略在一个全局的级别中工作。我们不把fetch="join"当作对集合映射的一种常用优化；你始终很少需要完全被初始化的集合。除了导致更高的内存消耗之外，每一个被外部联结的集合也都导致了更严重的笛卡儿积问题，我们很快就会对它进行更深入的讨论。

在实践中，你最可能在映射元数据中给bids集合启用一个批量的或者子查询策略。如果一个特定的过程（比如这个）需要内存中每一个Item的所有bids，你修改初始的HQL或者Criteria查询，并应用一种动态的抓取策略：

```
List<Item> allItems =
    session.createQuery("from Item i left join fetch i.bids")
           .list();

List<Item> allItems =
    session.createCriteria(Item.class)
        .setFetchMode("bids", FetchMode.JOIN)
        .list();

// Iterate through the collections...
```

这两个查询通过一个OUTER JOIN，为所有Item实例都生成了获取bids的单个SELECT（就像如果你已经用fetch="join"映射了这个集合时发生的那样）。

这可能是你第一次见到如何定义一个非全局的抓取策略。你放在映射元数据中的全局抓取计划和抓取策略设置只是始终应用的全局默认。任何优化过程也都需要更细粒度的规则，即只适用于特定的过程或者用例的抓取策略和抓取计划。下一章将更详细地讨论使用HQL和Criteria的抓取。你现在需要知道的就是有这些选项的存在。

使用延迟的集合时，n+1查询问题更频繁地出现。未被初始化的代理公开了相同的行为：你可能需要许多个SELECT来初始化正在特定过程中使用的所有对象。我们已经介绍过的优化指导方针是一样的，但是有一点不同：在<many-to-one>或者<one-to-one>关联中设置的fetch="join"是一般的优化，就像@ManyToOne(fetch=FetchType.EAGER)注解（Java Persistence中的默认）一样。单端关联的即时联结抓取，不同于集合的即时外部联结抓取，不会产生笛卡儿积问题。

2. 笛卡儿积问题

与n+1查询问题相反的是抓取过多数据的SELECT语句。如果试图抓取几个“并行的”集合，少不了会出现这个笛卡儿积问题。

假设你已经决定把一个全局的fetch="join"设置应用到Item的一个bids集合（尽管我们建议只在必要时才使用全局的预抓取和动态联结抓取策略）。Item类有其他的集合（例如images）。也假设你决定每件货品的所有图片都必须即时通过fetch="join"策略始终被加载：

```
<class name="Item">
    ...
    <set name="bids"
         inverse="true"
         fetch="join">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
    <set name="images"
         fetch="join">
        <key column="ITEM_ID"/>
        <composite-element class="Image">...
    </set>
</class>
```

如果通过一个即时的外部联结抓取策略映射两个并行的集合（它们自己的实体也一样），并加载所有的Item对象，Hibernate就执行一个创建这两个集合的乘积的SQL SELECT：

```
select item.*, bid.*, image.*
  from ITEM item
    left outer join BID bid on item.ITEM_ID = bid.ITEM_ID
    left outer join ITEM_IMAGE image on item.ITEM_ID = image.ITEM_ID
```

看看这个查询的结果集，如图13-6所示。

select item.*, bid.*, image.* from ITEM item
left outer join BID bid on item.ITEM_ID = bid.ITEM_ID
left outer join ITEM_IMAGE image on item.ITEM_ID = image.ITEM_ID

ITEM_ID	DESCRIPTION	...	BID_ID	ITEM_ID	AMOUNT	...	IMAGE_NAME
1	Item Nr. One	...	1	1	99.00	...	foo.jpg
1	Item Nr. One	...	1	1	99.00	...	bar.jpg
1	Item Nr. One	...	2	1	100.00	...	foo.jpg
1	Item Nr. One	...	2	1	100.00	...	bar.jpg
1	Item Nr. One	...	3	1	101.00	...	foo.jpg
1	Item Nr. One	...	3	1	101.00	...	bar.jpg
2	Item Nr. Two	...	4	2	4.99	...	baz.jpg
3	Item Nr. Three	...	NULL	NULL	NULL	...	NULL

图13-6 乘积是带有许多行的两个外部联结的结果

这个结果集包含大量多余的数据。1号货品有3个出价和2张图片，2号货品有1个出价和1张图片，3号货品没有出价和图片。这个乘积的大小取决于正在获取的集合的大小：3次2，1次1，再加1，共8个结果行。现在想象你在数据库中有1 000件货品，每件货品有20个出价和5张图片——你会看到一个约有100 000行的结果集！这个结果的大小可能有好几兆。在数据库中创建这个结果集需要的处理时间不仅长，而且需要的内存也相当大。所有数据都必须通过网络转移。当Hibernate把结果集封送到持久化对象和集合的时候，它立即移除所有重复——跳过多余的信息。3个查询当然更快了！

如果用`fetch="subselect"`映射这个并行的集合，就会得到3个查询；这就是对并行集合所建议的优化。然而，每一种规则都有例外。只要集合很小，乘积就可能是可以接受的抓取策略。注意，通过外部联结`SELECT`即时抓取的并行的单值关联一般不会产生乘积。

最后，虽然Hibernate让你通过`fetch="join"`在2个（或者甚至更多个）并行的集合中创建笛卡儿积，但是如果你试图在并行的`<bag>`集合上启用`fetch="join"`，它就会抛出异常。乘积的结果集无法被转化为包集合，因为Hibernate无法知道哪些行包含有效的重复（包允许重复）以及哪些行不包含。如果使用包集合（它们是Java Persistence中默认的`@OneToMany`集合），就不启用导致乘积的抓取策略。给包集合的并行即时抓取使用子查询或者立即的二级查询抓取。

全局和动态的抓取策略帮助你解决n+1查询和笛卡儿积问题。Hibernate提供另一个选项来初始化一个有时候有用的代理或者集合。

3．强制初始化代理和集合

代理或者集合包装器，每当调用它的任何方法时都被自动初始化（除了标识符属性获取方法之外，它可以返回标识符值，而不抓取底层的持久化对象）。预抓取和即时联结抓取都是获取你需要的所有数据的可能的解决方案。

你有时候想要在脱管状态下使用对象网络。获取所有应该脱管的对象和集合，然后关闭持久化上下文。

在这个场景中，在关闭持久化上下文之前显式地初始化一个对象有时候有用，而不用求助于改变全局的抓取策略或者不同的查询（我们认为这是你应该始终首选的解决方案）。

可以使用静态的方法`Hibernate.initialize()`给代理进行手工初始化：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item item = (Item) session.get(Item.class, new Long(1234));

Hibernate.initialize( item.getSeller() );

tx.commit();
session.close();

processDetached( item.getSeller() );
...
```

`Hibernate.initialize()`可以传递一个集合包装器或者代理。注意，如果你把一个集合包装器传递到`initialize()`，它不会初始化这个集合所引用的目标实体对象。在前一个示例中，`Hibernate.initialize(item.getBids())`不加载该集合内部的所有`Bid`对象。它用`Bid`对象的

代理初始化该集合！

几乎没有必要用这个静态的辅助方法来显式地初始化；你应该始终首选使用HQL或者Criteria的动态抓取。

既然你知道了所有的选项、问题和可能性，让我们来完成一个典型的应用程序优化过程。

4．逐步优化

首先，启用Hibernate SQL日志。你也应该准备针对特定的数据库Schema而阅读、理解和评估SQL查询及其性能特征：单个外部联结操作比两个查询更好吗？所有的索引都使用得当吗？以及什么是数据库内部的高速缓存命中率？让数据管理员帮助你做性能评估；只要他具备关于最好的SQL执行计划的知识。（如果想变成这个领域的专家，我们推荐这本书：Dan Tow著的*SQL Tunning* [Tow，2003]。）

这两个配置属性hibernate.format_sql和hibernate.use_sql_comments，使得在日志文件中阅读和分类SQL语句变得容易多了。优化期间启用这两个。

接下来，逐个执行你应用程序的用例，并注意Hibernate执行了多少个以及哪些SQL语句。用例可以是Web应用程序的单个屏幕，或者是一系列用户对话框。这个步骤也涉及了收集你在每个用例中使用的对象获取方法：通过对象链接，通过标识符、HQL和Criteria查询获取。你的目标是通过调优元数据中的默认抓取计划和抓取策略，把每个用例的SQL语句数量（和复杂性）降下来。

现在该定义抓取计划了。默认情况下的一切都是延迟加载的。考虑转换到多对一、一对一和（有时）集合映射中的lazy="false"（或者FetchType.EAGER）。全局的抓取计划定义了始终被即时加载的对象。优化你的查询，并且如果你需要即时（不是全局）地加载对象，就启用即时抓取，但要在特定的过程（仅有的一个用例）中。

一旦定义了抓取计划，并且知道一个特定的用例所需要的数据量，就优化这个数据获取的方式。你可能遇到两个常见的问题：

- SQL语句使用太复杂且太慢的联结操作。首先和你的数据库管理员一起优化SQL执行计划。如果这样还没有解决问题，就移除集合映射中的fetch="join"（或者先不设置它）。通过考虑它们是真正需要fetch="join"策略，还是被关联的对象应该通过二级查询进行加载，来最优化你的多对一和一对一关联。也可以通过全局的hibernate.max_fetch_depth配置选项来尝试调优，但是记住这个值最好保持在1～5之间。
- 可能执行过多的SQL语句。在多对一和一对一关联映射中设置fetch="join"。在极少数情况下，如果你绝对确定，就启用fetch="join"，禁用对特定集合的延迟加载。记住，每个持久化类都有不止一个被即时抓取的集合会导致乘积。利用批量或者子查询，来评估你的用例是否能从集合的预抓取中受益。使用3～15之间的批量大小。

设置了一种新的抓取策略之后，重新运行用例，并再次检查生成的SQL。注意这些SQL语句，并进行下一个用例。优化了所有用例之后，再次检查每一个用例，看看是否任何全局优化对于其他的会有副作用。凭借一些经验，你将第一次就能够很容易地避免任何负面的影响，并得到想要的结果。

这种优化技术不仅仅对于默认的抓取策略实用；也可以用它调优HQL和Criteria查询，它们可以动态地定义抓取计划和抓取策略。你经常能够用一个新的动态查询或者现有查询的一种变形，替换全局的抓取设置——关于这些选项，下一章将进行更加深入的讨论。

下一节介绍Hibernate高速缓存系统。在应用层中高速缓存数据，是你能够在任何复杂的多用户应用程序中使用的一种补充优化。

13.3　高速缓存基本原理

我们之所以认为使用对象/关系持久层远胜于用直接的JDBC创建的应用程序，一个重要的理由在于：高速缓存的潜能。虽然我们对大多数的应用程序在不使用高速缓存的情况下应该可能实现满意的性能有严重的分歧，但对于某些类型的应用程序，尤其对于主要用于读取的应用程序或者在数据库中保存重大元数据的应用程序来说，这是毫无疑义的，因为高速缓存会对性能造成巨大的影响。此外，把一个高并发的应用程序扩展到上千个在线事务，通常需要一些高速缓存来减少（多个）数据库服务器中的负荷。

我们通过一些背景信息开始对高速缓存的探讨。这包括阐述不同的高速缓存和同一性范围，以及高速缓存在事务隔离性中的影响。这种信息和这些规则一般来说可以应用到高速缓存，并且不仅仅只对Hibernate应用程序有效。这个讨论给你提供了一个背景，让你理解为什么Hibernate高速缓存系统是这样的。然后介绍Hibernate高速缓存系统，并教你如何启用、调优和管理一级和二级Hibernate高速缓存。建议你在开始使用高速缓存之前，认真学习本节中安排的基本原理。没有这些基础，你可能很快遇到难以调试的并发问题和数据完整性的危险。

高速缓存就是性能优化，因此它自然不是Java Persistence或者EJB 3.0规范的一部分。每一个供应商都对优化提供不同的解决方案，尤其是任何二级高速缓存。本节中介绍的所有策略和选项都适用于原生的Hibernate应用程序或者依赖Java Persistence接口并使用Hibernate作为持久化提供程序的应用程序。

高速缓存使当前数据库状态的表示接近应用程序，要么在内存中，要么在应用程序服务器机器的磁盘上。高速缓存是数据的一个本地副本，处在应用程序和数据库之间，可以用来避免数据库命中，每当：

- 应用程序通过标识符（主键）执行查找的时候。
- 持久层延迟解析一个关联或者集合的时候。

也可能高速缓存查询的结果。如你将在第15章所见，高速缓存查询结果的性能收获在许多案例中都十分微小，因此这项功能很少使用。

在讨论Hibernate的高速缓存如何工作之前，先快速学习一下不同的高速缓存选项，并看看它们如何相关到同一性和并发。

13.3.1　高速缓存策略和范围

高速缓存的概念在对象/关系持久化中是如此基本，以致如果你不先知道它使用了哪（几）

种高速缓存策略，就无法理解性能、可伸缩性或者ORM实现的事务语义。有3种主要的高速缓存类型：

- 事务范围高速缓存——添加到当前的工作单元，它可能是一个数据库事务，甚至是一个对话。它只有在工作单元运行时才是有效的，才能被使用。每一个工作单元都有自己的高速缓存。这个高速缓存中的数据不会被并发访问。
- 过程范围高速缓存——在许多（可能并发的）工作单元或者事务之间共享。这意味着过程范围高速缓存中的数据被并发运行的线程访问，显然隐含着事务隔离性。
- 集群范围高速缓存——在同一台机器的多个过程之间或者一个集群中的多台机器上共享。这里，网络通信是一个值得考虑的关键点。

过程范围高速缓存可以把持久化实例本身保存在高速缓存中，或者可以用分解的格式只保存它们的持久化状态。然后，每个访问共享高速缓存的工作单元都从被高速缓存的数据中重新组合一个持久化实例。

集群范围高速缓存需要某种远程过程通信（remote process communication）来维持一致性。高速缓存信息必须被复制到集群中的所有节点。对于许多（并非所有的）应用程序来说，集群范围高速缓存的价值并不太确定，因为读取和更新高速缓存可能只是在一定程度上比直接到数据库快一些。

持久层可以提供多级高速缓存。例如，事务范围中的高速缓存丢失（cache miss，在高速缓存中查找一个项目，但是它并没有被包含在该高速缓存中）可能接着在过程范围中查找。数据库请求是最后的办法。

持久层使用的高速缓存类型影响着对象同一性的范围（Java对象同一性和数据库同一性之间的关系）。

1. 高速缓存和对象同一性

考虑事务范围的高速缓存。这个高速缓存似乎很自然地也被用作对象的同一性范围。这意味着高速缓存实现同一性处理：对于使用相同数据库标识符的对象的两个查找返回了实际上相同的Java实例。因此，如果持久化机制也提供工作单元范围的对象同一性，事务范围高速缓存就是理想的。

利用过程范围高速缓存的持久化机制可能选择实现过程范围的同一性。在这种情况下，对象同一性就相当于整个过程的数据库同一性。在两个并发运行的工作单元中，使用相同的数据库标识符的两个查找产生了相同的Java实例。另一种方法是，从过程范围的高速缓存中获取的对象可能按值（by value）被返回。此时，每个工作单元获取它自己的状态副本（考虑原始数据），并且生成的持久化实例也不相同。高速缓存的范围和对象同一性的范围就不再相同。

集群范围高速缓存始终需要远程通信，并且在面向POJO的持久化解决方案（如Hibernate）的案例中，对象始终按值被远程传递。因此，集群范围高速缓存无法保证跨越集群的同一性。

对于典型的Web或者企业应用程序的架构来说，对象同一性的范围被限制为单个工作单元最方便。换句话说，在两个并发的线程中有相同的对象既不必要也不理想。在其他种类的应用程序中（包括有些桌面型的或者胖客户端的架构），使用过程范围的对象同一性可能很恰当。这在内

存极端受限的应用程序中尤其合适——工作单元范围高速缓存的内存消耗与并发线程的数量成正比。

然而，过程范围同一性的真正缺点在于，需要同步对高速缓存中持久化实例的访问，它极可能导致死锁，以及由于锁争用而减少了可伸缩性。

2．高速缓存和并发

任何允许多个工作单元共享相同持久化实例的ORM实现，都必须提供某些对象级锁定的形式，以确保并发访问的同步。这通常结合使用读/写锁（保存在内存中）与死锁侦测共同实现。像Hibernate这样的实现，给每一个工作单元（工作单元范围的同一性）维护一个独特的实例组，很大程度地避免了这些问题。

我们的观点是，应该避免在内存中保存锁，至少对于多用户可伸缩性高于一切的Web应用程序和企业应用程序应该是如此。在这些应用程序中，通常无需通过并发的工作单元比较对象同一性；每个用户都应该完全与其他的用户隔离开来。

当底层的关系数据库实现一个多版本的并发模型（如Oracle或者PostgreSQL）时，这种观点有一个特别明显的案例。给对象/关系持久化高速缓存重新定义事务语义或者底层数据库的并发模型，这有点不像我们想要的结果。

再次考虑选项。如果你也使用工作单元范围的对象同一性，并且如果对于高并发的多用户系统来说它是最佳策略，那么事务/工作单元范围的高速缓存就是首选。这个一级高速缓存是强制的，因为它也保证同一的对象。然而，这并非是你可以使用的唯一高速缓存。对于有些数据，二级高速缓存被界定到按值返回数据的过程（或者集群）中可能会有好处。因此这个场景有两个高速缓存层；稍后你会见到Hibernate使用这种方法。

来讨论哪种数据受益于二级高速缓存——换句话说，除了强制的一级事务范围的高速缓存之外，什么时候打开过程（或者集群）范围的二级高速缓存。

3．高速缓存和事务隔离性

过程或者集群范围高速缓存使得从一个工作单位中的数据库获取到的数据对另一个工作单元可见。这对事务隔离性可能有一些讨厌的负面影响。

首先，如果应用程序有对数据库的非专有访问，就不应该使用过程范围高速缓存，除了对很少改变且可以被高速缓存期限安全地刷新的数据之外。这种数据类型经常出现在内容管理类型的应用程序中，但很少在EIS或者财务应用程序中出现。

对于非专有的访问，要小心两个主要的场景：

- 被集群的应用程序；
- 共享的遗留数据。

任何被设计为可伸缩的应用程序，都必须支持集群的操作。过程范围高速缓存不维持集群中不同机器上不同高速缓存之间的一致性。此时，应该使用集群范围（分布式的）二级高速缓存而不是过程范围高速缓存。

许多Java应用程序都通过其他的应用程序共享对它们数据库的访问。既然如此，你就不应该使用任何超出工作单元范围一级高速缓存的高速缓存类型。高速缓存系统无法知道遗留应用程序

什么时候更新了共享的数据。实际上，虽然在改变数据库时有可能实现应用级的功能来触发过程（或者集群）范围高速缓存的失效，但我们不知道实现这一点的任何标准的或者最佳的方法。的确，它从来就不是Hibernate的内建特性。如果你实现这样一种解决方案，很可能是你自己努力的结果，因为它特定于所使用的环境和产品。

在考虑了非专有的数据访问之后，你应该创建应用程序数据需要的隔离级别了。并非每一个高速缓存实现都遵循所有的事务隔离级别，并且找出需要的级别很重要。我们来看一下从过程（或者集群）范围高速缓存中最受益的数据。在实践中，当我们进行这一评估时，发现依赖数据模型图（或者类图）很有帮助。在图上标注，说明一个特定的实体（或者类）对于二级高速缓存是好还是不好的备选对象。

完全的ORM解决方案让你单独给每个类配置二级高速缓存。高速缓存的好的备选类是表示以下数据的类：

- 很少改变的数据；
- 不重要的数据（如内容管理数据）；
- 应用程序固有的而非共享的数据。

对于二级高速缓存来说，不好的备选类是：

- 经常被更新的数据；
- 财务数据；
- 通过遗留应用程序共享的数据。

通常应用的规则不是唯一的。许多应用程序都有大量包含下列属性的类：

- 少量的实例；
- 被另一个类或者几个类的许多实例引用的每一个实例；
- 很少（或者从不）更新的实例。

这种数据有时称作*引用数据*（reference data）。引用数据的例子是邮政编码、参考地址、办公地点、静态文本消息等。引用数据对于利用过程或者集群范围的高速缓存而言是极好的备选对象，并且如果这个数据被高速缓存，任何大量使用引用数据的应用程序都将受益匪浅。你允许数据在高速缓存超时到期时刷新。

前几节通过工作单元范围的一级和可选的二级过程或者集群范围高速缓存，讲述了双层高速缓存系统。这很接近Hibernate的高速缓存系统了。

13.3.2 Hibernate 高速缓存架构

就像我们前面提示过的，Hibernate有一个两级的高速缓存架构。这个系统的各种元素如图13-7所示：

- 一级高速缓存是持久化上下文高速缓存。一个Hibernate `Session`的寿命相当于单个请求（通常用一个数据库事务实现）或者单个对话。这是个强制的一级高速缓存，它保证对象的范围和数据库同一性（例外的是`StatelessSession`，它没有持久化上下文）。

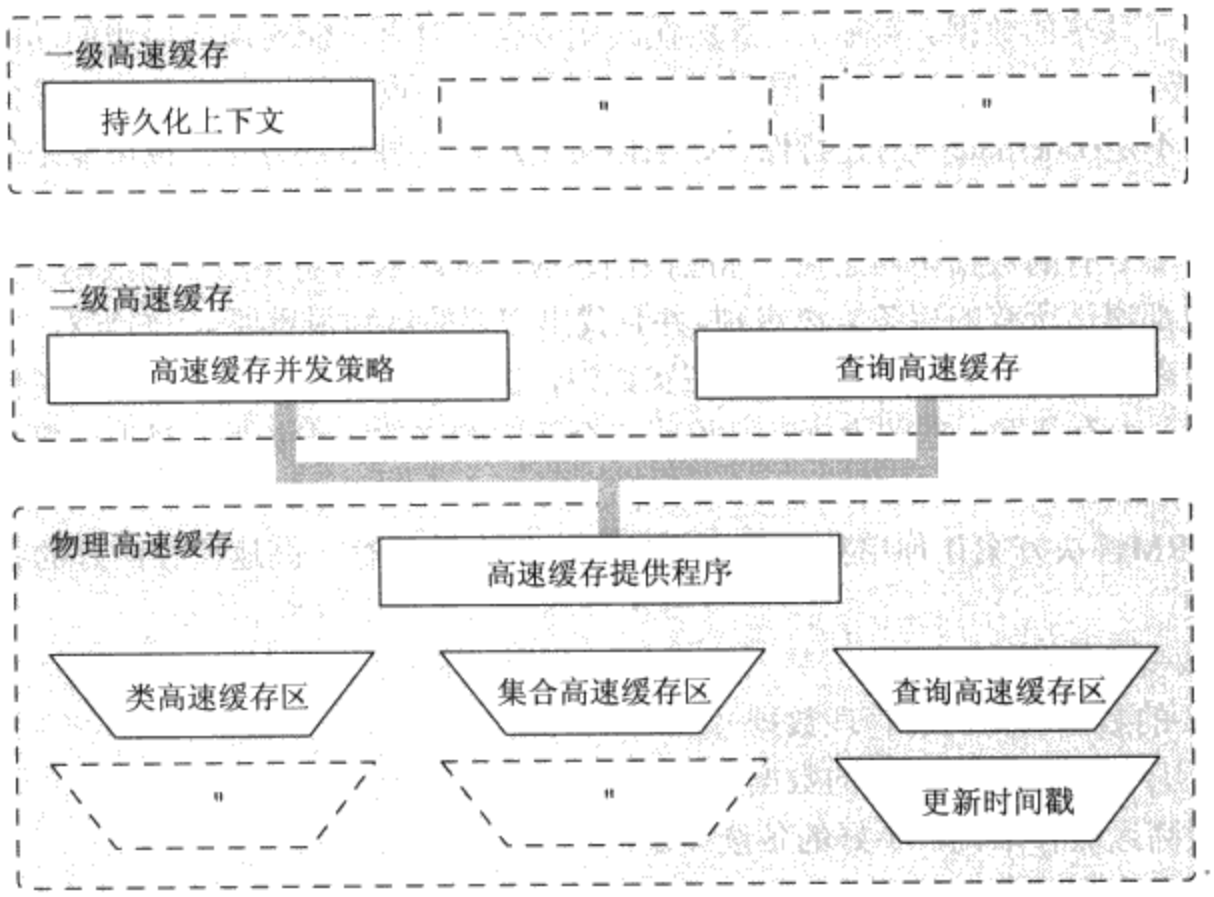

图13-7　Hibernate的二级高速缓存架构

- Hibernate中的二级高速缓存是可插拔的，并且可以被界定到过程或者集群。这是一个状态的高速缓存（按值返回），而不是实际的持久化实例的高速缓存。高速缓存并发策略给特定的数据项目定义了事务隔离细节，而高速缓存提供程序表示了物理高速缓存实现。二级高速缓存的使用是可选的，可以在按类和按集合的基础上配置——每一个这样的高速缓存都利用它自己的物理高速缓存区域。
- Hibernate也给与二级高速缓存密切整合的查询结果集实现高速缓存。这是个可选的特性；它需要两个额外的物理高速缓存区域，当表最后一次被更新时保存被高速缓存的查询结果和时间戳。我们在接下来的章节中讨论查询高速缓存，因为它的使用与正在执行的查询密切相关。

我们已经深入讨论过一级高速缓存、持久化上下文。现在直接进入可选的二级高速缓存。

1．Hibernate二级高速缓存

Hibernate的二级高速缓存有过程或者集群范围：已经从特定的`SessionFactory`开始的（或者与特定持久化单元的`EntityManager`关联的）所有持久化上下文共享同一个二级高速缓存。

持久化实例以分解的形式保存在二级高速缓存中。把分解当作一个过程，有点像序列化（但是这个算法比Java序列化要快很多）。

这个过程/集群范围高速缓存的内部实现没有什么值得关注的。更为重要的是高速缓存策略（cache policy）的正确使用——高速缓存策略和物理高速缓存提供程序。

不同种类的数据需要不同的高速缓存策略：读取与写入之比不同，数据库表的大小就不同，并且有些表与其他外部应用程序共享。二级高速缓存在一个单独的类或者集合角色的粒度中是可

配置的。例如，这样让你为引用数据类启用二级高速缓存，并对表示财务记录的类禁用它。这个高速缓存策略涉及如下设置：

- 二级高速缓存是否被启用；
- Hibernate并发策略；
- 高速缓存过期策略（如超时、LRU和内存敏感性）；
- 高速缓存的物理格式（内存、被索引的文件、集群复制）。

并非所有的类都受益于高速缓存，因此能够禁用二级高速缓存很重要。重申一下，高速缓存通常只对主要用来读取的类有用。如果你有更新比读取更经常的数据，就不要启用二级高速缓存，即使所有其他的高速缓存条件都符合！更新期间维护高速缓存的代价可能远远超出更快读取的性能受益。此外，二级高速缓存在与其他的写入应用程序共享数据库的系统中可能很危险。如前几节中所述，你在这里必须给想要启用高速缓存的每个类和集合进行小心的判断。

Hibernate二级高速缓存的创建分两步。首先，必须决定使用哪种*并发策略*（concurrency strategy）。之后，利用*高速缓存提供程序*配置高速缓存过期和物理高速缓存属性。

2. 内建的并发策略

并发策略是一个媒介：它负责在高速缓存中保存数据的项目，并从高速缓存中获取它们。这是个重要的角色，因为它也为特定的项目定义了事务隔离语义。如果想要启用二级高速缓存，就必须给每个持久化类和集合决定使用哪种并发策略。

下面4个内建的并发策略表示递减的事务隔离方面的严格级别：

- 事务（transactional）——只可用于托管环境，如有必要，它还保证完全的事务隔离直到*可重复读取*（repeatable read）。给主要用于读取的数据使用这种策略，因为在这种数据中，防止并发事务中的废弃数据最为关键，极少数情况下用于更新。
- 读/写（read-write）——这种策略利用时间戳机制，维护*读取提交*（read committed）隔离，并且只在非集群环境中可用。还是给主要用于读取的数据使用这种策略，因为在这种数据中防止并发事务中的废弃数据最为关键，极少数情况下用于更新。
- 非严格读/写（nonstrict-read-write）——不提供高速缓存和数据库之间的一致性保证。如果有可能并发访问相同的实体，你应该配置一个足够短的超时期限。否则，则可能从高速缓存中读取废弃的数据。如果数据几乎不变（几个小时、几天或者甚至一周），并且废弃的数据不可能是关键的关注点，那就使用这种策略。
- 只读（read-only）——并发策略适合于从不改变的数据。它只用于引用数据。

注意，递减的严格性带来了性能的递增。在把它用在产品中之前，必须小心评估包含完整事务隔离的集群高速缓存的性能。在许多情况下，如果绝不考虑废弃的数据，可能最好给特定的类禁用二级高速缓存！首先以禁用了二级高速缓存的应用程序为基准。为好的备选类启用它，一次一个，同时继续测试系统的可伸缩性，并评估并发策略。

通过实现`org.hibernate.cache.CacheConcurrencyStrategy`，有可能定义自己的并发策略，但这是个相对困难的任务，并且只适合于少数的优化案例。

考虑了要给高速缓存的备选类使用的并发策略之后，下一步就是挑选*高速缓存提供程序*。这

个提供程序是一个插件，是高速缓存系统的物理实现。

3．选择高速缓存提供程序

现在，Hibernate强制你给整个应用程序选择单个的高速缓存提供程序。Hibernate内建了以下开源产品的提供程序：

- EHCache是特意用于单个JVM中简单的过程范围高速缓存的高速缓存提供程序。它可以高速缓存在内存或者磁盘中，并支持可选的Hibernate查询结果高速缓存。（EHCache的最新版本现在支持被集群的高速缓存，但我们还没有对它进行过测试。）
- OpenSymphony OSCache是一项服务，它通过一组丰富的过期策略和查询高速缓存支持，支持在单个JVM中高速缓存到内存和磁盘。
- SwarmCache是基于JGroups的集群高速缓存。它使用集群的失效，但不支持Hibernate查询高速缓存。
- JBoss Cache是一个完全事务复制的集群高速缓存，也基于JGroups多播库（multicast library）。它支持复制或者失效、同步或者不同步的通信，以及乐观锁和悲观锁。支持Hibernate查询高速缓存，假设时钟在集群中同步。

通过实现`org.hibernate.cache.CacheProvider`给其他产品编写一个适配器很容易。许多商业高速缓存系统都可以通过这个接口插入到Hibernate。

并非每个高速缓存提供程序都可以与每一种并发策略兼容！表13-1中的兼容性矩阵将帮助你选择一种适当的组合。

表13-1　高速缓存并发策略支持

并发策略高速缓存提供程序	只　读	非严格读/写	读/写	事　务
EHCache	×	×	×	
OSCache	×	×	×	
SwarmCache	×	×		
JBoss Cache	×			×

创建高速缓存涉及两个步骤：首先，在映射元数据中看看持久化类和集合，并决定要给每个类和每个集合使用哪种高速缓存并发策略。然后，在全局的Hibernate配置中启用首选的高速缓存提供程序，并定制提供程序特定的设置和物理高速缓存区域。例如，如果正在使用OSCache，就编辑oscache.properties，或者给EHCache编辑类路径中的ehcache.xml。

来给CaveatEmptor的`Category`、`Item`和`Bid`类启用高速缓存。

13.4　高速缓存实践

首先，考虑每个实体类和集合，并找出哪种高速缓存并发策略可能适合。在给本地的和集群的高速缓存选择好高速缓存提供程序之后，将编写它们的（几个）配置文件。

13.4.1　选择并发控制策略

`Category`只有少量的实例并且很少更新，而且实例在许多用户之间共享。它是供二级高速缓

存使用的一个很好的备选类。

从添加Hibernate高速缓存Category实例所需的映射元素开始：

```
<class name="auction.model.Category"
       table="CATEGORY">

    <cache usage="read-write"/>

    <id ...

</class>
```

usage="read-write"属性告诉Hibernate，给auction.model.Category高速缓存使用一个读/写并发策略。每当导航到Category，或者当按标识符加载Category时，Hibernate现在就命中二级高速缓存。

如果使用注解，就需要Hibernate扩展：

```
@Entity
@Table(name = "CATEGORY")
@org.hibernate.annotations.Cache(usage =
    org.hibernate.annotations.CacheConcurrencyStrategy.READ_WRITE
)
public class Category { ... }
```

你用read-wirte代替了nonstrict-read-write，因为Category是一个高并发的类，在许多并发事务中共享。（很显然，读取提交隔离级别就足够好了。）nonstrict-read-write只依赖高速缓存过期（超时），但你更喜欢让对类别所做的改变立即可见。

类高速缓存始终对持久化类的整个层次结构而被启用。你无法只高速缓存一个特定子类的实例。

这个映射足以告诉Hibernate去高速缓存所有简单的Category属性值，而不是被关联的实体或者集合的状态。集合需要它们自己的<cache>区域。给items集合使用read-write并发策略：

```
<class name="auction.model.Category"
       table="CATEGORY">

    <cache usage="read-write"/>

    <id ...

    <set name="items">
        <cache usage="read-write"/>
        <key ...
    </set>

</class>
```

集合高速缓存的区域名称是完全匹配的类名加上集合属性的名称：auction.model.Category.items。@org.hibernate.annotations.Cache注解也可以在集合字段或者获取方法中进行声明。

调用aCategory.getItems()时，这个高速缓存设置生效——换句话说，集合高速缓存是一个包含“哪些项目处在哪些类别中”的区域。它是只有标识符的高速缓存，在该区域中没有实际的Category或者Item数据。

如果你需要Item实例本身被高速缓存，就必须启用Item类的高速缓存。读/写（read-write）策略尤其适合。用户可不想根据可能失效的Item数据做出决定（例如，发出一个出价）。更进一步，考虑bids的集合：bids集合中的特定Bid是不可变的，但是bids的集合是可变的，并且并发的工作单元需要毫不延迟地看到集合元素的添加或者移除：

```
<class name="Item"
       table="ITEM">

    <cache usage="read-write"/>

    <id ...

    <set name="bids">
        <cache usage="read-write"/>
        <key ...
    </set>

</class>
```

给Bid类应用只读（read-only）策略：

```
<class name="Bid"
       table="BID" mutable="false">

    <cache usage="read-only"/>

    <id ...

</class>
```

因此，Bid数据在高速缓存中从来不会过期，因为它只能被创建而从不更新。（出价当然可能被高速缓存提供程序终止——例如，如果达到了高速缓存中对象的最多数量时。）如果Bid实例被删除，Hibernate也从高速缓存中移除数据，但是不对这种做法提供任何事务的保证。

User是可以通过非严格读/写（nonstrict-read-wirte）策略而被高速缓存的类的一个示例，但我们不确定它对于高速缓存用户是否有意义。

下面来设置高速缓存提供程序，设置它的过期策略和高速缓存的物理区域。使用高速缓存区域单独地配置类和集合高速缓存。

13.4.2　理解高速缓存区域

Hibernate在不同的高速缓存区域（cache region）中保存不同的类/集合。区域是一个具名的高速缓存：这个句柄使你可以通过它在高速缓存提供程序配置中引用类和集合，并设置适用于该区域的过期策略。一种更为图形化的描述是，区域是一桶桶的数据，它们有两种类型：一种区域类型包含实体实例的分解数据，另一种类型只包含通过集合而被链接的实体标识符。

对于类高速缓存而言，区域的名称是类名；对于集合高速缓存而言，是类名加属性名。Category实例被高速缓存在具名auction.model.Category的区域中，而items集合则被高速缓存在具名auction.model.Category.items的区域中。

Hibernate具名hibernate.cache.region_prefix的配置属性可能被用来给特定的SessionFactory或者持久化单元指定区域名前缀。例如，如果前缀设置为dbl，Category就被

高速缓存在具名dbl.auction.model.Category的区域中。如果应用程序使用多个SessionFactory实例或者持久化单元，这项设置就是必需的。没有它，不同持久化单元的高速缓存区域名称就可能冲突。

既然了解了高速缓存区域，就可以配置auction.model.Category高速缓存的物理属性了。首先来选择一种高速缓存提供程序。假设你正在单个JVM中运行拍卖应用程序，那么你不需要集群感知的提供程序。

13.4.3 设置本地的高速缓存提供程序

需要设置选择了高速缓存提供程序的配置属性：

```
hibernate.cache.provider_class = org.hibernate.cache.EhCacheProvider
```

在这个例子中选择EHCache作为二级高速缓存。

现在，你需要指定高速缓存区域的属性。EHCache有自己的配置文件ehcache.xml，在应用程序的类路径中。Hibernate发行包中为所有绑定的高速缓存提供程序捆绑了示例配置文件，因此我们建议你阅读那些文件中的使用注释，看看详细的配置，并给我们没有明确提到的所有选项假设默认值。

echache.xml中用于Category类的高速缓存配置可能看起来像这样：

```
<cache name="auction.model.Category"
       maxElementsInMemory="500"
       eternal="true"
       timeToIdleSeconds="0"
       timeToLiveSeconds="0"
       overflowToDisk="false"
/>
```

有少量的Category实例。因此，你通过选择一个大于系统中类别数量的高速缓存大小限制、并设置eternal="true"、通过超时禁用清除。没有必要通过超时终止被高速缓存的数据，因为Category高速缓存并发策略是读/写，并且因为没有其他的应用程序直接在数据库中改变类别数据。你还禁用了基于磁盘的高速缓存溢出，因为你知道Category的实例很少，因此内存消耗不会有问题。

另一方面，Bid很小并且不可变，但是有很多，因此你必须配置EHCache来小心管理高速缓存内存消耗。你把过期超时和最大高速缓存大小限制都用上了：

```
<cache name="auction.model.Bid"
       maxElementsInMemory="50000"
       eternal="false"
       timeToIdleSeconds="1800"
       timeToLiveSeconds="100000"
       overflowToDisk="false"
/>
```

timeToIdleSeconds属性定义了以秒为单位的从一个元素最后一次在高速缓存中被访问开始的过期时间。你必须在这里设置一个有意义的值，因为你不想让没有用到的出价消耗内存。

timeToLiveSeconds属性定义了以秒为单位的从元素被添加到高速缓存中开始的最大过期时间。由于出价是不可变的，因此如果它们正被正常访问，你就不需要把它们从高速缓存中移除。因而，timeToLiveSeconds被设置成一个很大的数值。

结果，当高速缓存总大小达到最大限制50 000个元素时，那么在最近的30分钟内没有被使用或者最近使用频度最小的被高速缓存的出价就会从高速缓存中被移除。

在这个示例中，你禁用了基于磁盘的高速缓存，因为你预计应用程序服务器会被部署为与数据库相同的机器。如果所期待的物理架构不同，则可能启用基于磁盘的高速缓存来减少网络通信量。在本地磁盘上访问数据比跨过网络访问数据库更快。

如你所见，最佳的高速缓存清除策略是特定于数据和应用程序。必须考虑许多外部的因素，包括应用程序服务器机器中可用的内存、希望在数据库机器上的加载、网络延迟、遗留应用程序的存在，等等。这些因素中有一些不可能在开发时得知，因此经常需要在产品环境中或者模拟的环境中反复调试不同设置的性能影响。我们考虑通过二级高速缓存优化一些在开发期间你不会去做的事情，因为在没有实际的数据集和并发的情况下的测试，并不能代表系统的最终性能和可伸缩性。这在集群机器中带有复制的高速缓存的更复杂场景中尤其如此。

13.4.4　设置重复的高速缓存

如果应用程序部署在单个虚拟机上，EHCache就是一个极好的高速缓存提供程序。然而，支持上千个并发用户的企业应用程序则可能需要更大的计算能力，并且可伸缩的应用程序对于项目的成功可能十分关键。Hibernate应用程序天生就是可伸缩的：Hibernate没有任何方面限制应用程序要部署在哪些节点上。通过稍微改变高速缓存配置，甚至就可能使用一个集群的高速缓存系统了。

我们推荐JBoss Cache，一个基于TreeCache和JGroups多播库的集群安全的高速缓存系统。JBoss Cache是非常可伸缩的，并且集群通信可以用任何可能的方式进行调优。

现在进入对CaveatEmptor的JBoss Cache配置，一个两节点（称作节点A和节点B）的小集群。然而，我们只对这个话题做肤浅的探讨，集群配置天生复杂，并且许多设置取决于特定的场景。

首先，必须检查所有使用只读或者事务作为高速缓存并发策略的所有映射文件。这些是JBoss Cache提供程序唯一支持的策略。有一种很好的技巧有助于避免这种搜索，并在未来替换问题。不要在映射文件中放置<cache>元素，而是集中在hibernate.cfg.xml中的高速缓存配置：

```
<hibernate-configuration>
<session-factory>

    <property .../>
    <mapping .../>

    <class-cache
            class="org.hibernate.auction.model.Item"
            usage="transactional"/>

    <collection-cache
            collection="org.hibernate.auction.model.Item.bids"
            usage="transactional"/>
```

```
</session-factory>

</hibernate-configuration>
```

在这个例子中，你给Item和bids集合启用了事务高速缓存。然而，有一个重要的警告：在编写本书之时，如果你在Item的映射文件中也有<cache>元素，Hibernate就遇到了冲突。因此，你不能使用全局的配置覆盖映射文件设置。建议从一开始就使用集中的高速缓存配置，尤其当你不确定应用程序会被如何部署的时候。通过单个配置位置调优高速缓存设置也更加容易。

集群配置的下一步是JBoss Cache提供程序的配置。首先，在Hibernate配置中启用它——例如，如果你没有在hibernate.cfg.xml中使用属性：

```
<property name="cache.provider_class">
    org.hibernate.cache.TreeCacheProvider
</property>
```

JBoss Cache有它自己的配置文件treecache.xml，在应用程序的类路径中。在有些场景中，集群中的每个节点都需要不同的配置，并且你必须确保部署时把正确的文件复制到类路径中去。来看一个典型的配置文件。在一个两节点的集群（具名MyCluster）中，这个文件用在节点A上：

```
<server>

  <classpath codebase="./lib"
             archives="jboss-cache.jar, jgroups.jar"/>

  <mbean code="org.jboss.cache.TreeCache"
      name="jboss.cache:service=TreeCache">

<depends>jboss:service=Naming</depends>
<depends>jboss:service=TransactionManager</depends>

<attribute name="TransactionManagerLookupClass">
    org.jboss.cache.GenericTransactionManagerLookup
</attribute>

<attribute name="ClusterName">MyCluster</attribute>

<attribute name="NodeLockingScheme">PESSIMISTIC</attribute>
<attribute name="CacheMode">REPL_SYNC</attribute>
<attribute name="IsolationLevel">REPEATABLE_READ</attribute>

<attribute name="FetchInMemoryState">false</attribute>
<attribute name="InitialStateRetrievalTimeout">20000</attribute>
<attribute name="SyncReplTimeout">20000</attribute>
<attribute name="LockAcquisitionTimeout">15000</attribute>

<attribute name="ClusterConfig">
    <config>
      <UDP loopback="false"/>

      <PING timeout="2000"
            num_initial_members="3"
            up_thread="false"
            down_thread="false"/>

      <FD_SOCK/>

      <pbcast.NAKACK gc_lag="50"
```

```
                        retransmit_timeout="600,1200,2400,4800"
                        max_xmit_size="8192"
                        up_thread="false" down_thread="false"/>
                <UNICAST timeout="600,1200,2400"
                         window_size="100"
                         min_threshold="10"
                         down_thread="false"/>
                <pbcast.STABLE desired_avg_gossip="20000"
                               up_thread="false"
                               down_thread="false"/>
                <FRAG frag_size="8192"
                      down_thread="false"
                      up_thread="false"/>
                <pbcast.GMS join_timeout="5000"
                            join_retry_timeout="2000"
                            shun="true" print_local_addr="true"/>
                <pbcast.STATE_TRANSFER up_thread="true"
                                       down_thread="true"/>
            </config>
        </attribute>
    </mbean>
</server>
```

当然，这个配置文件乍看可能有点吓人，但它很容易理解。你必须知道，它不仅仅是JBoss Cache的一个配置文件，它还集多种角色于一身：JBoss AS部署的JMX服务配置、TreeCache的配置文件、JGroups的细粒度配置和通信库。

忽略与JBoss部署相关的前面几行，看看第一个属性`TransactionManagerLookupClass`。`GenericTransactionManagerLoopup`试图在最流行的应用程序服务器中找到事务管理器，但它也适用于没有JTA的独立环境（没有事务管理器的集群高速缓存很少见）。如果JBoss Cache在启动时抛出异常，告诉你它无法找到事务管理器，你就必须自己给JPA提供程序/应用程序服务器创建这样一个查找类。

接下来，是使用同步通信的被复制高速缓存的配置属性。这意味着发送同步消息的节点等待着，直到群组中的所有节点都获得这条消息。对于真正的被复制高速缓存来说，这是一个很好的选择；如果节点B是一个热备份（hot standby，如果节点A失败，立即取而代之的一个节点）而不是一个积极的搭档的话，异步不阻塞的通信将更合适。这是一个故障转移（failover）与计算能力的问题，两者都是配置集群的好理由。大多数的配置属性都应该一目了然，比如节点进入集群时的超时和状态抓取。

JBoss Cache也可以清除元素，防止内存耗尽。在这个例子中，你不配置清除策略，因此高速缓存慢慢开始填充所有可用的内存。对于清除策略配置，你必须参考JBoss Cache文档；Hibernate区域名称的用法和清除设置类似于EHCache。

JBoss Cache也支持失效，但不支持集群中被修改数据的复制，这是一种可能更好的执行选择。然而，Hibernate查询高速缓存需要复制。你也可以转换到OPTIMISTIC锁而不是悲观锁，再次加

强集群高速缓存的可伸缩性。这么做需要另一种Hibernate高速缓存提供程序插件org.hibernate.cache.OptimisticTreeCacheProvider。

最后，看看JGroups集群通信配置。通信协议的顺序极为重要，因此不要随便改变或者增加行。最值得关注的是第一个协议<UDP>。如果节点A是一台Microsoft Windows机器（在这个例子中不是），loopback属性就必须设置为true。

其他的JGroups属性更复杂，可以在JGroups文档中找到。它们处理发现算法（discovery algorithm），用于在群组中侦测新节点、失败侦测，以及一般来说还有群组通信的管理。

在改变持久化类的高速缓存并发策略为事务（或者只读），并为节点A创建了treecache.xml文件之后，就可以启动应用程序并观察日志输出了。建议给org.jboss.cache包启用DEBUG日志；然后你就会看到JBoss Cache如何读取配置，并报告节点A为集群中的第一个节点。为了部署节点B，你要在该节点上部署应用程序；不需要改变配置文件（如果第二个节点也不是Microsoft Windows机器）。如果启动这第二个节点，你应该在这两个节点上都看到联结消息。现在Hibernate应用程序在集群中使用完全事务的高速缓存。

还有最后一项可选的设置要考虑。对于集群高速缓存提供程序而言，把Hibernate配置选项hibernate.cache.use_minimal_puts设置为true可能更好。当启用这项设置时，Hibernate只在检查以确保该项目还没有被高速缓存之后才把它添加到高速缓存中。如果高速缓存写入（存）比高速缓存读取（取）更贵，则是这种策略执行得更好。这是针对集群中被复制高速缓存的案例，而不是针对本地高速缓存或者依赖失效（而非复制）的高速缓存提供程序。

无论使用集群还是本地高速缓存，你有时候都需要通过编程来控制它，不论为了测试，还是为了调优。

13.4.5 控制二级高速缓存

Hibernate有一些有用的方法，可以帮助你测试和调优高速缓存。给二级高速缓存hibernate.cache.use_second_level_cache考虑全局的配置转换。默认情况下，映射文件（在hibernate.cfg.xml或者注解）中的任何<cache>元素都触发二级高速缓存，并在启动时加载高速缓存提供程序。如果想要全局地禁用二级高速缓存，而不移除高速缓存映射元素或者注解，那么就设置这个配置属性为false。

就像Session和EntityManager提供通过编程来控制持久化上下文一级高速缓存的方法一样，SessionFactory对二级高速缓存也一样。在JPA应用程序中，必须访问底层的内部SessionFactory，如2.2.4节中所述。

可以调用evict()，通过指定类和对象标识符值，从二级高速缓存中移除元素：

```
SessionFactory.evict( Category.class, new Long(123) );
```

也可以通过指定一个区域名称，来清除一个特定类的所有元素，或者清除一个特定集合角色：

```
SessionFactory.evict("auction.model.Category");
```

你将很少需要用到这些控制机制。还要注意二级高速缓存的清除是非事务的，也就是说，高速缓存区域在清除期间没有被锁定。

Hibernate还提供CacheMode选项，它可以由特定的Session启用。想象你想要在Session中把许多个对象插入到数据库。需要以批量来完成这项工作，以避免内存耗尽——每一个对象都被添加到一级高速缓存中。然而，如果为实体类启用了它，它也被添加到二级高速缓存中。CacheMode控制Hibernate与二级高速缓存的交互：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

session.setCacheMode(CacheMode.IGNORE);

for ( int i=0; i<100000; i++ ) {
    Item item = new Item(...);
    session.save(item);
    if ( i % 100 == 0 ) {
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

设置CacheMode.IGNORE告诉Hibernate不要在这个特定的Session中与二级高速缓存交互。可用的选项如下：

- CacheMode.NORMAL——默认的行为。
- CacheMode.IGNORE——Hibernate从来不与二级高速缓存交互，除了更新发生时使被高速缓存的项目失效之外。
- CacheMode.GET——Hibernate可能从二级高速缓存中读取项目，但不添加项目，除了更新发生时使项目失效之外。
- CacheMode.PUT——Hibernate从来不从二级高速缓存中读取项目，但是当它从数据库中读取项目时，会把项目添加到高速缓存。
- CacheMode.REFRESH——Hibernate从来不从二级高速缓存中读取项目，但是当它从数据库中读取项目时，会把项目添加到高速缓存。在这种模式下，hibernate.cache.use_minimal_puts的作用被忽略，以便在一个复制的集群高速缓存中强制高速缓存刷新。

除了NORMAL和IGNORE之外，好的用例很少。

这样就结束了对Hibernate应用程序中一级和二级高速缓存的讨论。我们想要重申一项在本节一开始时提到过的声明：你的应用程序不用二级高速缓存也应该执行得令人满意。如果你通过启用二级高速缓存，使应用程序中的一个特定过程在2秒而不是50秒内运行，你就只是治了标，而没有治本。抓取计划和抓取策略的定制始终是你的第一个优化步骤；然后，通过二级高速缓存使应用程序更迅速，并把它扩展到它将必须在产品中处理的并发事务加载。

13.5 小结

本章创建了一个全局的抓取计划，并定义了哪些对象和集合应该始终被加载到内存中。你定义了基于用例的抓取计划，以及想要如何访问被关联的实体，以及如何在应用程序中遍历集合。

接下来，你给抓取计划选择了正确的抓取策略。目标是把SQL语句的数量和必须执行的每个SQL语句的复杂性减到最小。你尤其想要通过各种优化策略，来避免我们详细阐述过的n+1查询和笛卡儿积问题。

本章后半部分通过高速缓存背后的理论介绍了高速缓存，并提供了一张清单，可以用它来找出哪些类和集合对于Hibernate的可选二级高速缓存是好的备选对象。然后，通过本地的EHCache提供程序和集群启用的JBoss Cache，给一些类和集合配置和启用了二级高速缓存。

表13-2展现了可以用来比较原生的Hibernate特性和Java Persistence的概括。

表13-2 第13章中Hibernate和JPA的对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate支持通过代理或者基于拦截的延迟加载定义抓取计划	Hibernate通过代理或者基于拦截的延迟加载实现Java Persistence提供程序
Hibernate允许细粒度地控制抓取计划和抓取策略	Java Persistence给抓取计划声明标准化了注解，Hibernate扩展用于细粒度的抓取策略优化
Hibernate提供一个可选的二级类和集合数据高速缓存（配置在Hibernate配置文件或者XML映射文件中）	在实体和集合中，给高速缓存并发策略的声明使用Hibernate注解

接下来的章节专门处理查询，并介绍如何利用所有的Hibernate和Java Persistence接口来编写和执行HQL、JPA QL、SQL和Criteria查询。