Масштабируемая веб-архитектура и распределенные системы

Федосеева А.А.

Распределенная система - система, в которой обработка информации сосредоточена не на одной вычислительной машине, а распределена между несколькими компьютерами

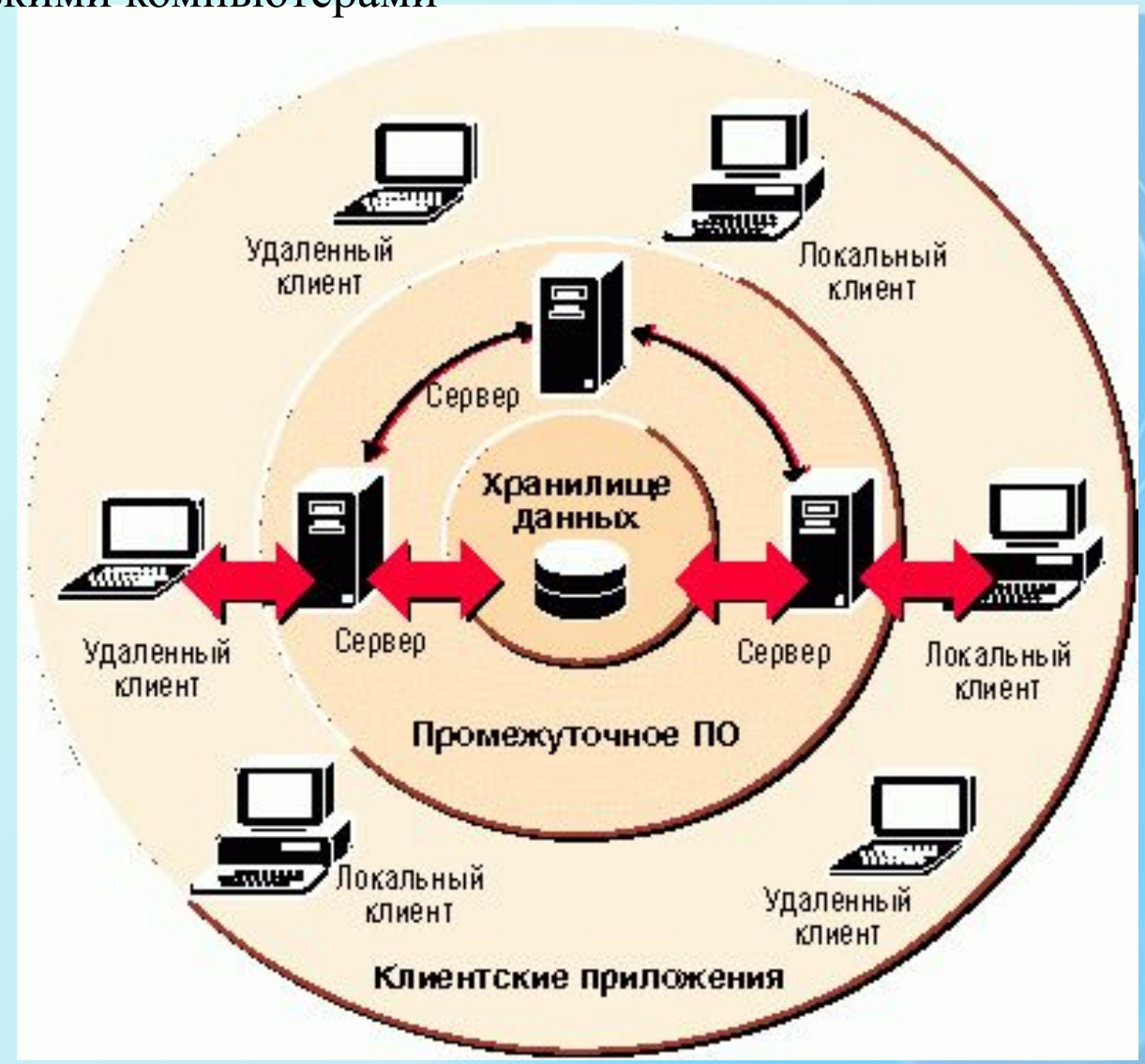

Схема распределенных вычислений

Выделяется три типа архитектур распределенных систем.

- *Архитектура клиент/сервер*. В этой модели систему можно представить как набор сервисов, предоставляемых серверами клиентам. В таких системах серверы и клиенты значительно отличаются друг от друга.
- *Трехзвенная архитектура*. В этой модели сервер предоставляет клиентам сервисы не напрямую, а посредством сервера бизнес-логики.
- *Архитектура распределенных объектов*. В этом случае между серверами и клиентами нет различий и систему можно представить как набор взаимодействующих объектов, местоположение которых не имеет особого значения. Между поставщиком сервисов и их пользователями не существует различий

Недостатки распределенных систем

- *Сложность*. Намного труднее понять и оценить свойства распределенных систем в целом, их сложнее проектировать, тестировать и обслуживать. Также производительность системы зависит от скорости работы сети, а не отдельных процессоров. Перераспределение ресурсов может существенно изменить скорость работы системы.
- *Безопасность*. Обычно доступ к системе можно получить с нескольких разных машин, сообщения в сети могут просматриваться и перехватываться. Поэтому в распределенной системе намного труднее поддерживать безопасность.
- *Управляемость*. Система может состоять из разнотипных компьютеров, на которых могут быть установлены различные версии операционных систем. Ошибки на одной машине могут распространиться непредсказуемым образом на другие машины.
- *Непредсказуемость*. Реакция распределенных систем на некоторые события непредсказуема и зависит от полной загрузки системы, ее организации и сетевой нагрузки. Так как эти параметры могут постоянно изменятся, поэтому время ответа на запрос может существенно отличаться от времени.

Принципы построения распределенных веб-систем

- **Доступность:** длительность работоспособного состояния веб-сайта критически важна по отношению к репутации и функциональности многих компаний. Для некоторых более крупных онлайновых розничных магазинов, недоступность даже в течение нескольких минут может привести к тысячам или миллионам долларов потерянного дохода. Таким образом, разработка их постоянно доступных и эластичных к отказу систем и является и фундаментальным деловым и технологическим требованием. Высокая доступность в распределенных системах требует внимательного рассмотрения избыточности для ключевых компонентов, быстрого восстановления после частичных системных отказов и сглаженного сокращения возможностей при возникновении проблем.
- **Производительность:** Производительность веб-сайта стала важным показателем для большинства сайтов. Скорость веб-сайта влияет на работу и удовлетворенность пользователей, а также ранжирование поисковыми системами — фактор, который непосредственно влияет на удержание аудитории и доход. В результате, ключом является создание системы, которая оптимизирована для быстрых ответов и низких задержек.
- **Надежность:** система должна быть надежной, таким образом, чтобы определенный запрос на получение данных единообразно возвращал определенные данные. В случае изменения данных или обновления, то тот же запрос должен возвращать новые данные. Пользователи должны знать, если что-то записано в систему или храниться в ней, то можно быть уверенным, что оно будет оставаться на своем месте для возможности извлечения данных впоследствии.

* Принципы построения распределенных веб-систем

- **Масштабируемость:** Когда дело доходит до любой крупной распределенной системы, размер оказывается всего лишь одним пунктом из целого списка, который необходимо учитывать. Не менее важным являются усилия, направленные на увеличение пропускной способности для обработки больших объемов нагрузки, которая обычно и именуется масштабируемость системы. Масштабируемость может относиться к различным параметрам системы: количество дополнительного трафика, с которым она может справиться, насколько легко нарастить ёмкость запоминающего устройства, или насколько больше других транзакций может быть обработано.
- **Управляемость:** проектирование системы, которая проста в эксплуатации еще один важный фактор. Управляемость системы приравнивается к масштабируемости операций «обслуживание" и «обновления». Для обеспечения управляемости необходимо рассмотреть вопросы простоты диагностики и понимания возникающих проблем, легкости проведения обновлений или модификации, прихотливости системы в эксплуатации. (То есть, работает ли она как положено без отказов или исключений?)
- **Стоимость:** Стоимость является важным фактором. Она, очевидно, может включать в себя расходы на аппаратное и программное обеспечение, однако важно также рассматривать другие аспекты, необходимые для развертывания и поддержания системы. Количество времени разработчиков, требуемое для построения системы, объем оперативных усилий, необходимые для запуска системы, и даже достаточный уровень обучения — все должно быть предусмотрено. Стоимость представляет собой общую стоимость владения

Архитектура распределенных приложений

В разных источниках приводятся различные варианты построения распределенных приложений. И все они имеют право на существование, ведь такие приложения решают самый широкий круг задач во многих предметных областях, а неудержимое развитие средств разработки и технологий подталкивает к непрерывному совершенствованию.

Тем не менее существует наиболее общая архитектура распределенного приложения, согласно которой оно разбивается на несколько логических слоев, уровней обработки данных. Приложения, как известно, предназначены для обработки информации, и здесь мы можем выделить три главнейшие их функции:

- · представление данных (пользовательский уровень). Здесь пользователи приложения могут просмотреть необходимые данные, отправить на выполнение запрос, ввести в систему новые данные или отредактировать их;
- · обработка данных (промежуточный уровень, middleware). На этом уровне сконцентрирована бизнес-логика приложения, осуществляется управление потоками данных и организуется взаимодействие частей приложения. Именно концентрация всех функций обработки данных и управления на одном уровне считается основным преимуществом распределенных приложений;
- · хранение данных (уровень данных). Это уровень серверов баз данных. Здесь расположены сами серверы, базы данных, средства доступа к данным, различные вспомогательные инструменты

Основные уровни трехзвенной архитектуры распределенного приложения

Нередко такую архитектуру называют трехуровневой или трехзвенной. И очень часто на основе этих "трех китов" создается структура разрабатываемого приложения. При этом всегда отмечается, что каждый уровень может быть дополнительно разбит на несколько подуровней. Например, пользовательский уровень может быть разбит на собственно пользовательский интерфейс и правила проверки и обработки вводимых данных.

Распределение бизнес-логики по уровням распределенного приложения:

Четырехуровневая распределенная архитектура

- · представление данных (пользовательский уровень);
- · правила бизнес-логики (уровень обработки данных);
- · управление данными (уровень управления данными);
- · хранение данных (уровень хранения данных)

EJB (Java Beans)

Технологию **EJB** (Enterprise Java Beans) можно рассматривать с двух точек зрения:

-как **фреймворк**,

С точки зрения **фреймворка** EJB - это технология, предоставляющая множество готовых решений (управление транзакциями, безопасность, хранение информации и т.п.) для вашего приложения.

-как **компонент**.

С точки зрения компонента **EJB** - это всего-лишь **надстройка** над POJO*-классом, описываемая с помощью **аннотации**. Существует три типа компонентов EJB:

session beans - используется для описания бизнесс-логики приложения

message-driven beans - так же используется для бизнесс-логики

entities - используется для хранения данных

Достоинства	**Недостатки**
Быстрое и простое создание	Плохая масштабируемость
Java-оптимизация	Трудность интегрирования с существующими приложениями
Кроссплатформенность	Отсутствие международной стандартизации
Динамическая загрузка компонентов-переходников	Встроенная безопасность

*простой Java-объект, не унаследованный от какого-то специфического объекта и не реализующий никаких служебных интерфейсов сверх тех, которые нужны для бизнес-модели

Основные архитектуры EJB

Существует 2 основные архитектуры при разработке enterprise-приложений:

- традиционная слоистая архитектура (traditional layered architecture)
- domain-driven design (DDD)

Обе эти архитектуры предполагают разделение приложения на функциональные **слои**, каждый из которых используется для решения задач определенного плана.

Традиционная слоистая архитектура предполагает разделение приложения на **4 базовых слоя**: слой презентации, слой бизнесс-логики, слой хранения данных и непосредственно слой самой базы данных. Обычно слой **презентации** реализуется через **web-приложение** (т.е. используя JSP, JSF, GWT и т.п.) или **web-сервис** (что дает возможность написания клиента, к примеру, на C#). В нем реализовано взаимодействие с пользователем: формы для получения запросов от пользователя и средства для предоставления ему запрошенной информации.

Слой **бизнесс-логики** является основой для enterprise-приложения. В нем описываются бизнесс-процессы, производится поиск, авторизация и множество других вещей. Слой бизнесс-логики использует механизмы слоя **хранения данных**. Чем отличается слой **хранения данных** и слой **базы данных**? Тем, что в первом описываются высокоуровневые объектно-ориентированные механизмы для работы с сущностями БД, в то время как второй - это и есть непосредственно база данных (Oracle, MySQL и т.п.)

Архитектура **DDD** предполагает, что объекты обладают бизнесс-логикой, а не являются простой репликацией объектов БД. Многие программисты не любят наделять объекты логикой и создают отдельный слой, называемый service layer или application layer. Он похож на слой бизнесс-логики традиционной слоистой архитектуры за тем лишь отличием, что он намного *тоньше.*

Типы компонентов EJB\

Message-driven beans

Так же как и session beans используются для **бизнесс-логики**. Отличие в том, что клиенты никогда не вызывают MDB напрямую. Обычно сервер использует MDB в **асинхронных** запросах.

- *Session bean* представляет собой EJB-компоненту, связанную с одним клиентом. ``Бины" этого типа, как правило, имеют ограниченный срок жизни (хотя это и не обязательно), и редко участвуют в транзакциях. В частности, они обычно не восстанавливаются после сбоя сервера. В качестве примера session bean можно взять ``бин", который живет в Web-сервере и динамически создает HTML-страницы клиенту, при этом следя за тем, какая именно страница загружена у клиента. Когда же пользователь покидает Web-узел, или по истечении некоторого времени, session bean уничтожается. Несмотря на то, что в процессе своей работы, session bean мог сохранять некоторую информацию в базе данных, его **предназначение** заключается все-таки не в отображении состояния или в работе с ``вечными объектами", а просто **в выполнении некоторых функций на стороне сервера от имени одного клиента.**

- *Entity bean*, наоборот, представляет собой компоненту, работающую с постоянной (persistent) информацией, хранящейся, например, в базе данных. Entity beans ассоциируются с элементами баз данных и могут быть доступны одновременно нескольким пользователям. Так как информация в базе данных является постоянной, то и entity beans живут постоянно, ``выживая'', тем самым, после сбоев сервера (когда сервер восстанавливается после сбоя, он может восстановить ``бин'' из базы данных). Например, entity bean может представлять собой строку какой-нибудь таблицы из базы данных, или даже результат операции SELECT. В объектно-ориентированных базах данных, entity bean может представлять собой отдельный объект, со всеми его атрибутами и связями.

Entities и Java Persistence API

Одним из главным достоинством EJB3 стал новый механизм работы с persistence - возможность автоматически сохранять объекты в реляционной БД используя технологию объектно-реляционного маппинга (ORM).
В контексте EJB3 persistence провайдер - это ORM-фреймворк, который поддерживает EJB3 Java Persistence API (JPA). JPA определяет стандарт для:
конфигурации маппинга сущностей приложения и их отображения в таблицах БД;

EntityManager API - стандартный API для CRUD (create, read, update, delete) операций над сущностями;

Java Persistence Query Language (JPQL) - для поиска и получения данных приложения;

Можно сказать, что **session beans** - это "**глаголы**" приложения, в то время как **entities** - это "**существительные**".

EntityManager - это инерфейс, который связывает класс сущности приложения и его представление в БД. EntityManager знает как нужно добавлять сущности в базу, одновлять их, удалять, а так предоставляет механизмы для настройки производительности, кэширования, транзакций и т.д.

JPQL - это похожий на SQL язык запросов.

Реализация

Нассмотрим реализацию этих сущностей. В EJB3 мы используем **POJO** (Plain Old Java Objects), **POJI** (Plain Old Java Interfaces) и **аннотации**. Если с первыми двумя всё понятно, то про аннотации стоить поговорить отдельно. Аннотация записывается так:
@<имя аннотации>(<список парамет-значение>)
Какие аннотации могут быть?
Stateless - говорит контейнеру, что класс будет stateless session bean. Для него контейнер обеспечит безопасность потоков и менеджмент транзакций. Дополнительно, вы можете добавить другие свойства, например прозрачное управление безопасностью и перехватчики событий;

- **Local** - относится к интерфейсу и говорит, что bean реализующий интерфейс доступен локально
- **Remote** - относится к интерфейсу и говорит, что bean доступен через RMI (Remote Method Invocation)
- **EJB** - применятеся в коде, где мы используем bean.
- **Stateful** - говорит контейнеру, что класс будет stateful session bean.
- **Remove** - опциональная аннотация, которая используется с stateful бинами. Метод, помеченный как Remove говорит контейнеру, что после его исполнения нет больше смысла хранить bean, т.е. его состояние сбрасывается. Это бывает критично для производительности.
- **Entity** - говорит контейнеру, что класс будет сущностью БД
- **Table**(name="...") - указывает таблицу для маппинга
- **Id**, **Column** - параметры маппинга
- **WebService** - говорит, что интерфейс или класс будет представлять вэб-сервис.

Перехватчики

При создании enterprise-приложений часто возникает необходимость записывать лог вызываемых методов (в целях отладки или для лога безопасности), а так же контролировать доступ пользователей к отдельным частям приложения. Для этого используются **перехватчики** - объекты, методы которых вызываются **автоматически** при **вызове** метода **EJB**-бина. Объект-перехватчик является POJO, за тем лишь исключением, что метод, который должен вызываться автоматически аннотируется @AroundInvoke, например:

public class MyLogger { @AroundInvoke public Object logMethodEntry(InvocationContext invocationContext) throws Exception { System.out.println("Entering methid: " + invocationContext.getMethod().getName()); return invocationContext.proceed(); } } Обратите внимание на **возвращаемое значение** и **параметр** функции. В данном случае мы говорим контейнеру, что после вызова перехватчика можно вызывать метод, который он перехватил (или следующий за ним перехватчик), однако мы могли сгенерировать исключение или не вызывать метод proceed() и тогда метод EJB-бина не выполнился бы.

Использовать перехватчики можно двумя путями: указать его применение через **аннтоации** для каждого класса или метода в отдельности или указать **перехватчик по-умолчанию** для определенных (или всех) бинов.

Чтобы перехватчик применился только к **определенному методу** перед декларацией метода можно написать аннотацию @Interceptors(MyLogger.class). Чтобы перехватчик работал для **всех методов** бина эту же аннотацию можно было бы написать перед декларацией класса, например: @Interceptors(MyLogger.class) @Stateless public class MyClass { ... } Можно указывать **несколько** перехватчиков, тогда их перечисляют через запятую, например:
@Interceptors(MyLogger1.class, MyLogger2.class) **Порядок вызова** перехватчиков никак нельзя задать через аннотации, но его можно изменять, если описывать их через дескриптор развертывания.

Один перехватчик описывается так:
* com.example.MyLogger Сначала вызывается перехватчик по-умолчанию, потом специфичный для класса, а за ним для метода. Если вы не хотите чтобы для вашего метода вызывался перехватчик по-умолчанию - используйте аннотацию @ExcludeDefaultInterceptors или @ExcludeClassInterceptors.

DCOM

- Distributed Component Object Model (DCOM) - программная архитектура, разработанная компанией Microsoft для распределения приложений между несколькими компьютерами в сети. Программный компонент на одной из машин может использовать DCOM для передачи сообщения (его называют удаленным вызовом процедуры) к компоненту на другой машине. DCOM автоматически устанавливает соединение, передает сообщение и возвращает ответ удаленного компонента.
- Для того чтобы различные фрагменты сложного приложения могли работать вместе через Internet, необходимо обеспечить между ними надежные и защищенные соединения, а также создать специальную систему, которая направляет программный трафик.
- Для решения этой задачи компания Microsoft создала распределенную компонентную объектную модель Distributed Component Object Model (DCOM), которая встраивается в операционные системы Windows NT 4.0 и Windows 98 и выше.
- Преимуществом DCOM является, по мнению Карен Буше, аналитика The Standish Group, значительная простота использования. Если программисты пишут свои Windows-приложения с помощью ActiveX (предлагаемого Microsoft способа организации программных компонентов), то операционная система будет автоматически устанавливать необходимые соединения и перенаправлять трафик между компонентами, независимо от того, размещаются ли компоненты на той же машине или нет.
- Способность DCOM связывать компоненты позволила Microsoft наделить Windows рядом важных дополнительных возможностей, в частности, реализовать сервер Microsoft Transaction Server, отвечающий за выполнения транзакций баз данных через Internet. Новая же версия COM+ еще больше упростит программирование распределенных приложений, в частности, благодаря таким компонентам, как базы данных, размещаемые в оперативной памяти.

Достоинства и недостатки DCOM

Достоинства	Недостатки
Независимость от языка	Сложность реализации
Динамический/статический вызов	Зависимость от платформы
Динамическое нахождение объектов	Нет именования через URL
Масштабируемость	Нет проверки безопасности на уровне выполнении
Открытый стандарт (контроль со тороны TOG)	ActiveX компонент
Множественность Windows-программистов	Отсутствие альтернативных разработчиков

DCOM является лишь частным решением проблемы распределенных объектных систем. Он хорошо подходит для Microsoft-ориентированных сред. Как только в системе возникает необходимость работать с архитектурой, отличной от Windows, DCOM перестает быть оптимальным решением проблемы.

CORBA

- CORBA специфицирует инфраструктуру взаимодействия компонент (объектов) на представительском уровне и уровне приложений модели OSI. Она позволяет рассматривать все приложения в распределенной системе как объекты. Причем объекты могут одновременно играть роль и клиента, и сервера: роль клиента, если объект является инициатором вызова метода у другого объекта; роль сервера, если другой объект вызывает на нем какой-нибудь метод. Объекты-серверы обычно называют "реализацией объектов". Практика показывает, что большинство объектов одновременно исполняют роль и клиентов, и серверов, попеременно вызывая методы на других объектах и отвечая на вызове извне. Используя CORBA, тем самым, имеется возможность строить гораздо более гибкие системы, чем системы клиент-сервер, основанные на двухуровневой и трехуровневой архитектуре.

Достоинства и недостатки *CORBA*

Достоинства

1.Платформенная независимость

2.Языковая независимость

3.Динамические вызовы

4.Динамическое обнаружение объектов

5.Масштабируемость

6.CORBA-сервисы

7.Широкая индустриальная поддержка

Недостатки

1.Нет передачи параметров `по значению'

2.Отсутствует динамическая загрузка компонент-переходников

3.Нет именования через URL

ORM

- **Object-relational mapping** *(рус. Объектно-реляционное отображение)* — это технология программирования, которая позволяет преобразовывать несовместимые типы моделей в ООП, в частности, между хранилищем данных и объектами программирования. ORM используется для упрощения процесса сохранения объектов в реляционную базу данных и их извлечения, при этом ORM сама заботится о преобразовании данных между двумя несовместимыми состояниями. Большинство ORM-инструментов в значительной мере полагаются на метаданные базы данных и объектов, так что объектам ничего не нужно знать о структуре базы данных, а базе данных — ничего о том, как данные организованы в приложении. ORM обеспечивает полное разделение задач в хорошо спроектированных приложениях, при котором и база данных, и приложение могут работать с данными каждый в своей исходной форме.

Принцип работы ORM

Ключевой особенностью ORM является отображение, которое используется для привязки объекта к его данным в БД. ORM как бы создает «виртуальную» схему базы данных в памяти и позволяет манипулировать данными уже на уровне объектов. Отображение показывает как объект и его свойства связанны с одной или несколькими таблицами и их полями в базе данных. ORM использует информацию этого отображения для управления процессом преобразования данных между базой и формами объектов, а также для создания SQL-запросов для вставки, обновления и удаления данных в ответ на изменения, которые приложение вносит в эти объекты.

Преимущества и

1) Использование ORM в проекте избавляет разработчика от необходимости работы с SQL и написания большого количества кода, часто однообразного и подверженного ошибкам. Весь генерируемый ORM код предположительно хорошо проверен и оптимизирован, поэтому не нужно в целом задумывается о его тестировании.

Недостатки использования

1) потеря производительности. Это происходит потому, что большинство ORM предназначены для обработки широкого спектра сценариев использования данных, гораздо большего, чем любое отдельное приложение когда-либо сможет использовать.

Вопрос о целесообразности использования ORM по большому счету затрагивается только в больших проектах, которые сталкиваются с высокой нагрузкой, здесь приходится выбирать что более приоритетно — удобство или производительность?

Конечно, работа с БД посредством грамотно написанного SQL-кода будет намного эффективнее, но не стоит забывать и о таком параметре, как время — то, что с легкостью пишется с использованием ORM за неделю, можно реализовывать ни один месяц собственными усилиями. Кроме того, большинство современных ORM позволяют программисту при необходимости самому задавать код SQL-запросов. Без сомнений, для небольших проектов использование ORM будет куда более оправдано, чем разработка собственных библиотек для работы с БД.