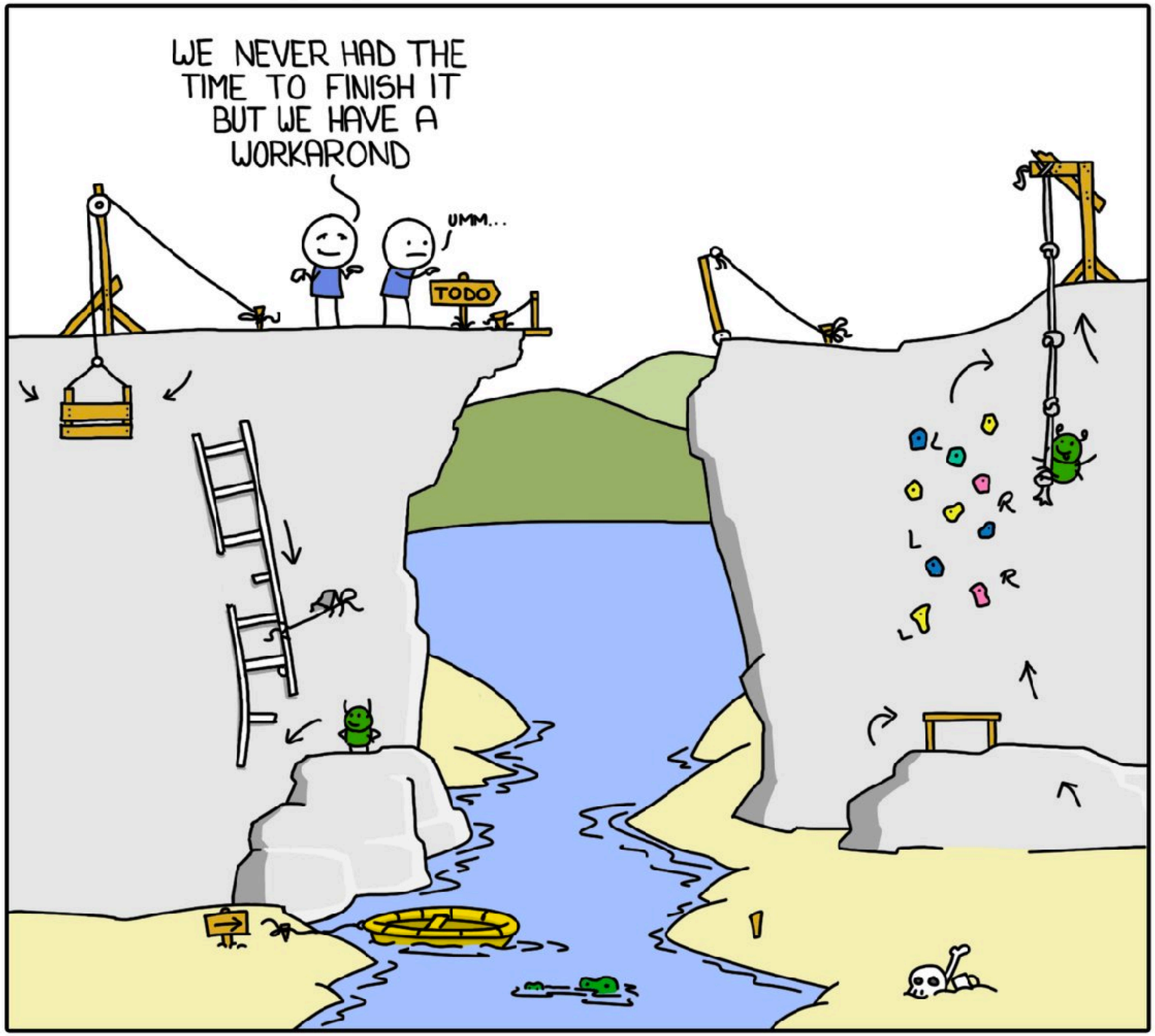
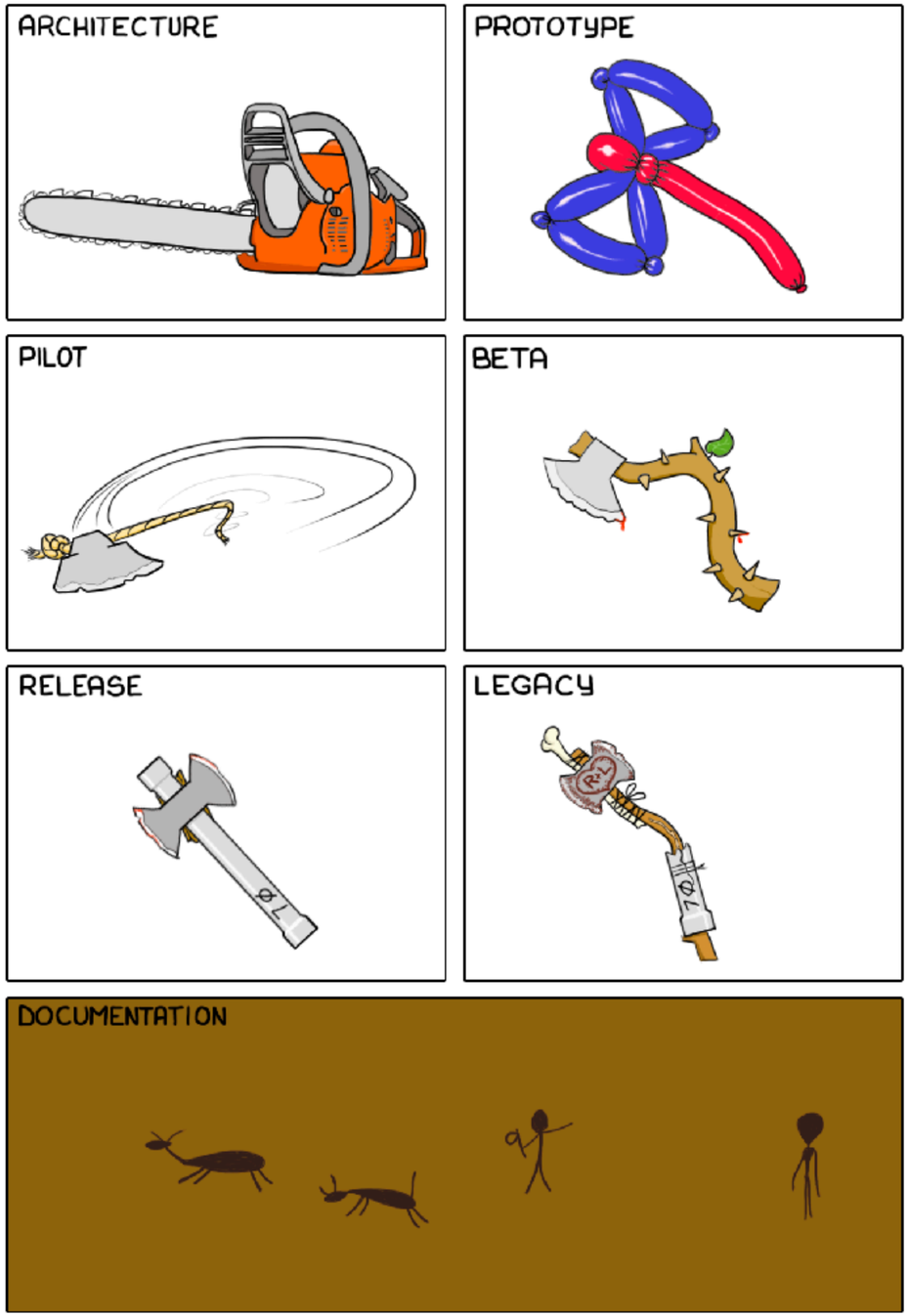
JAVA 达摩班

Spring Data JPA

WORKAROUND



CODE PROGRESSION



PostgreSQL 安装与使用 1

PostgreSQL

PostgreSQL是一个满足ACID事务处理的**对象关系数据库管理系统（ORDBMS）**，提供Linux，Unix和windows版本

- Atomicity：每个事务满足“所有或没有”
- Consistency：并行事务执行结果保持数据一致的状态
- Isolation：隔离在同一时间执行的事务，“感觉”只有每个事务自己在执行
- Duration：如果事务成功执行，那么状态会持久保存在数据库中

- **原生支持JSON格式**，可以作为MongoDB的，“No-SQL”的关系型数据库替代
- 简单而强大，高性能，高扩展性，处理海量数据和高负载网络应用能力（高并发用户）
- 流式操作，数据库模式，用户自定义对象（操作，数据类型，函数），嵌套事务，表继承，分区，特殊类型（Money，Geometry，IP，JSON，data range）
- 支持多种语言下的存储过程（Java，Python，Ruby，C/C++等）

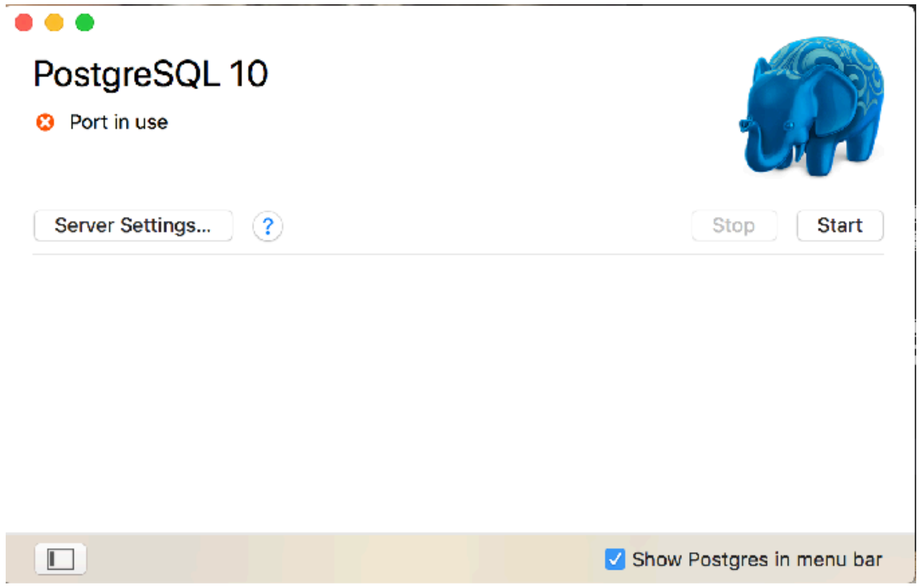
https://www.postgresql.org/docs/10/static/tutorial-sql.html
http://www.postgresqltutorial.com

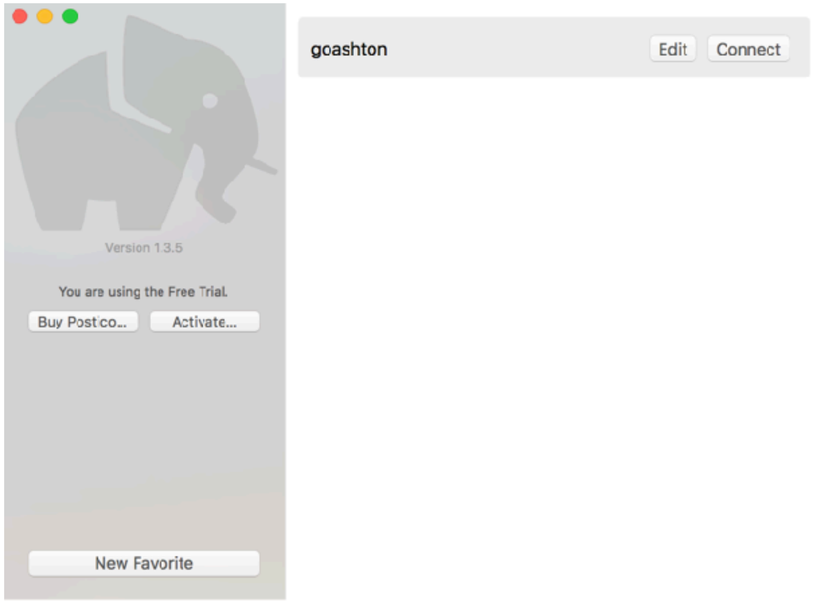
PostgreSQL 安装

数据库安装：

1. 图形化安装包：BigSQL， Postgres.app
2. 包管理工具：brew install postgresql

GUI管理界面：Postico，pgAdmin，Navicat





PostgreSQL

开机自启动

pg_ctl -D /usr/local/var/postgres start && brew services start postgresql

postgres -V

工具

- createuser 创建用户
- createdb 创建数据库
- dropuser 删除用户
- dropdb 删除数据库
- postgres: PostgreSQL server相关操作
- pg_dump: 将一个数据库内容导入文件
- pg_dumpall: 将所有数据库内容导入文件
- psql: 管理工具

PostgreSQL

psql是Postgres自带工具，可用非SQL方式管理数据库，表，用户等功能

```
  psql postgres
```

#查看用户，自动创建和登陆名一样的数据库用户

```
  \du
```

#退出

```
  \q
```

#创建用户 - CREATE ROLE 或 createuser

```
 CREATE ROLE tester WITH LOGIN PASSWORD ‘tester’;
 ALTER ROLE tester CREATEDB;
```

或

```
 createuser tester --createdb
```

```
postgres=# ALTER ROLE tester CREATEDB;
ALTER ROLE
postgres=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB                          | {}
 tester    | Create DB                                                  | {}
 xhji      | Superuser, Create role, Create DB, Replication             | {}
```

PostgreSQL

#创建数据库

psql postgres -U tester
CREATE DATABASE dharma_mall_app;
\list
\connect dharma_mall_app
\dt
或
createdb dharma_mall_app -U tester

#修改数据库名

ALTER DATABASE dharma_mall_app RENAME TO dharma_mall;

```
postgres=> CREATE DATABASE dharma_mall_app;
CREATE DATABASE
postgres=> GRANT ALL PRIVILEGES ON DATABASE dharma_mall_app TO tester;
GRANT
postgres=> \list
postgres=> \connect dharma_mall_app
You are now connected to database "dharma_mall_app" as user "tester".
dharma_mall_app=> \dt
No relations found.
dharma_mall_app=>
```

PostgreSQL

#创建表

```
create table product (
  id serial not null primary key,
  name varchar(20) not null,
  price decimal not null
);
```

#插入数据

```
insert into product (id, name, price) values
 (1, 'P20', 5999),
 (2, 'iPhoneX', 9999),
 (3, 'Mix2S', 3999),
 (4, 'R1', 8848);
```

#删除表

```
drop table product
```

```
dharma_mall_app=> \dt
         List of relations
 Schema |   Name   | Type  | Owner
--------+---------+-------+--------
 public | product | table | tester
(1 row)

dharma_mall_app=> \d+ product
                                                  Table "public.product"
 Column |         Type          |                      Modifiers                       | Storage  | Stats target | Description
--------+-----------------------+------------------------------------------------------+----------+--------------+-------------
 id     | integer               | not null default nextval('product_id_seq'::regclass) | plain    |              |
 name   | character varying(20) | not null                                             | extended |              |
 price  | numeric               | not null                                             | main     |              |
Indexes:
    "product_pkey" PRIMARY KEY, btree (id)

dharma_mall_app=> select column_name,data_type
dharma_mall_app-> from information_schema.columns
dharma_mall_app-> where table_name = 'product';
 column_name |     data_type
-------------+-------------------
 id          | integer
 name        | character varying
 price       | numeric
(3 rows)

dharma_mall_app=> select * from product;
 id |  name   | price
----+---------+-------
  1 | P20     |  5999
  2 | iPhoneX |  9999
  3 | Mix2S   |  3999
  4 | R1      |  8848
(4 rows)

dharma_mall_app=>
```

PostgreSQL备忘录 [1]

```
#以指定用户名登陆
psql -U [username];

#连接指定数据库
\c database_name;

#退出psql
\q

#列出当前数据库所有表
\l

#列出schema
\dn

#列出存储过程和函数
\df

#列出视图
\dv

#列出当前数据库表的信息
\dt
#列出当前数据库表更多的信息
\dt+

#查看表的详情（字段）
\d+ table_name

#查看表的字段，存储过程和函数
\df+ function_name

#打开/关闭格式美化
\x

#列出所有用户
\du

#创建新用户
CREATE ROLE role_name;

#创建用户和密码
CREATE ROLE username NOINHERIT LOGIN PASSWORD password;

#改变当前session中的角色
SET ROLE new_role;

#允许role_1设置自己的角色为role_2
GRANT role_2 TO role_1;
```

PostgreSQL备忘录 [2]

```
#创建数据库
CREATE DATABASE [IF NOT EXISTS] db_name;

#删除数据库
DROP DATABASE [IF EXISTS] db_name;

#创建表
CREATE [TEMP] TABLE [IF NOT EXISTS] table_name(
   pk SERIAL PRIMARY KEY,
   c1 type(size) NOT NULL,
   c2 type(size) NULL,
   ...
);

#添加列
ALTER TABLE table_name ADD COLUMN new_column_name TYPE;

#删除列
ALTER TABLE table_name DROP COLUMN column_name;

#重命名列
ALTER TABLE table_name RENAME column_name TO new_column_name;

#设置或删除列默认值
ALTER TABLE table_name ALTER COLUMN [SET DEFAULT value | DROP DEFAULT]
```

PostgreSQL备忘录 [3]

#添加主键
ALTER TABLE table_name ADD PRIMARY KEY (column,...);

#删除主键
ALTER TABLE table_name
DROP CONSTRAINT primary_key_constraint_name;

#表改名
ALTER TABLE table_name RENAME TO new_table_name;

#删除表，以及依赖对象
DROP TABLE [IF EXISTS] table_name CASCADE;

#创建视图
CREATE OR REPLACE view_name AS
query;

#创建迭代视图
CREATE RECURSIVE VIEW view_name(columns) AS
SELECT columns;

#创建物化视图
CREATE MATERIALIZED VIEW view_name
AS
query
WITH [NO] DATA;

#更新物化视图
REFRESH MATERIALIZED VIEW CONCURRENTLY view_name;

#删除视图
DROP VIEW [IF EXISTS] view_name;
DROP MATERIALIZED VIEW view_name;

#重命名视图
ALTER VIEW view_name RENAME TO new_name;
#创建索引
CREATE [UNIQUE] INDEX index_name
ON table (column,...)

#删除
DROP INDEX index_name;

PostgreSQL备忘录 [4]

#查询
SELECT * FROM table_name;

SELECT column, column2….
FROM table;

SELECT DISTINCT (column)
FROM table;

SELECT *
FROM table
WHERE condition;

SELECT column_1 AS new_column_1, ...
FROM table;

SELECT * FROM table_name
WHERE column LIKE '%value%'

SELECT * FROM table_name
WHERE column BETWEEN low AND high;

SELECT * FROM table_name
WHERE column IN (value1, value2,...);

SELECT * FROM table_name
LIMIT limit OFFSET offset
ORDER BY column_name;

#Join
SELECT *
FROM table1
INNER JOIN table2 ON conditions

SELECT *
FROM table1
LEFT JOIN table2 ON conditions

SELECT *
FROM table1
FULL OUTER JOIN table2 ON conditions

SELECT *
FROM table1
CROSS JOIN table2;

SELECT *
FROM table1
NATURAL JOIN table2;

PostgreSQL备忘录 [5]

```
SELECT COUNT (*)
FROM table_name;

SELECT column, column2, ...
FROM table
ORDER BY column ASC [DESC], column2 ASC [DESC],...;

SELECT *
FROM table
GROUP BY column_1, column_2, ...;
Filter groups using the HAVING clause.

SELECT *
FROM table
GROUP BY column_1
HAVING condition;
Set operations

SELECT * FROM table1
UNION [EXCEPT, INTERSECT]
SELECT * FROM table2;

#插入数据
INSERT INTO table_name(column1,column2,...)
VALUES(value_1,value_2,...),
      (value_1,value_2,…)...

#修改
UPDATE table
SET column_1 = value_1,
    ...
WHERE condition;

#删除
DELETE FROM table_name;

DELETE FROM table_name
WHERE condition;

#性能
#显示查询计划
EXPLAIN query;

#显示，执行查询计划
EXPLAIN ANALYZE query;

#收集统计数据
ANALYZE table_name;
```

从JDBC到JPA

2

JDBC和ORM

JDBC

是Java Database Connectivity的简写形式，Java SE定义的数据库连接标准
通过“状态”执行SQL语句，查询结果以“ResultSets”形式返回，支持事务，缓存等
主要用于关系型数据库，管理和数据库的连接（直接连接 + 连接池）
数据库开发商实现并提供各自的JDBC驱动库

ORM

是Object/Relational Mapping“的简写形式，映射Java POJO对象到关系型数据库中，是一种持久化类型
持久化是指存活在JVM应用之外的数据库对象
优点：

1.领域模型模式：开发者更多集中在业务概念，每个对象都是有业务意义的，而不是关心数据表结构
2.面向对象思想：可以按照面向对象思想组织对象，通过对象图查询而不是关系模型
3.提高开发速度&减少代码：自动映射JDBC ResultSet到POJO，减少CRUD操作的模版，减少同步数据变化的代码
4.便携性：ORM是独立于数据库的，通过OO API查询而不是直接的SQL，且自动生成厂商特定的SQL语句
5.性能优化：管理关系映射，支持并发，多租户，事务操作，灵活扩展性等

缺点：

1.速度慢
2.操作复杂，想象下EJB/XML

DAO模式

DAO是Data Access Object的简称，是一种**设计模式**，用于把数据持久化逻辑分离到一个单独的层，隔离service层和底层数据处理。

DAO = 模型 (业务模型，例如product，用于在层之间传输)
+ **接口** (操作model的接口)
+ **接口实现** (连接数据库，实现model操作)

- 服务层不关心数据来源，持久层DAO改变不影响服务层使用。例如从Mysql换成Postgresql，只需要改变DAO实现
- 低耦合，服务层调用DAO接口，解耦服务层和DAO实现
- 由于DAO实现（持久层）是完全独立实现，所以方便单元测试
- 符合“工作在接口，而不是实现”的OO原则

JPA

JPA是现代化的数据持久化技术规范，是实现ORM的Java标准API，定义了供ORM提供商实现的接口

> The Java Persistence API (JPA) is a Java application programming interface specification that describes the management of relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition.

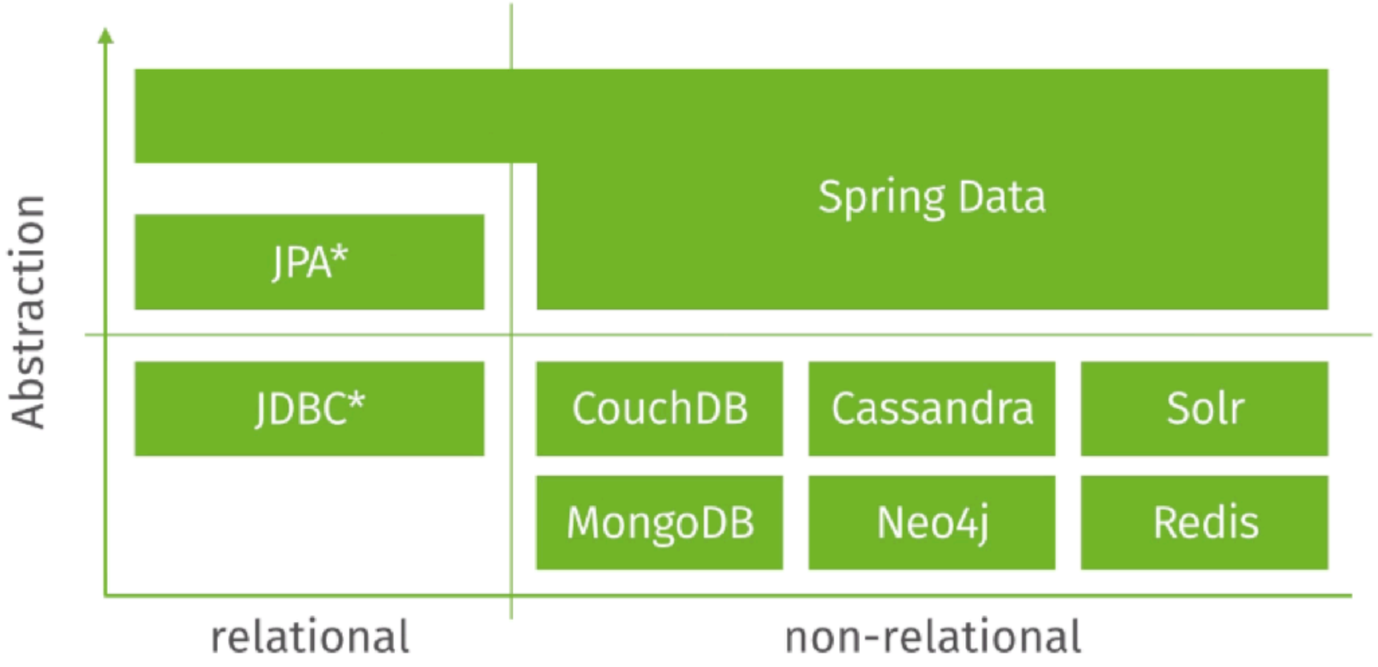
JPA最早包含在EJB 3.0规范中，由于entity bean使用复杂，并且只用于Java EE，所以被抽取出来成为独立的JPA规范，并且很快就有Hibernate，TopLink Essential支持并实现该规范，现在它同时被Java SE和EE支持

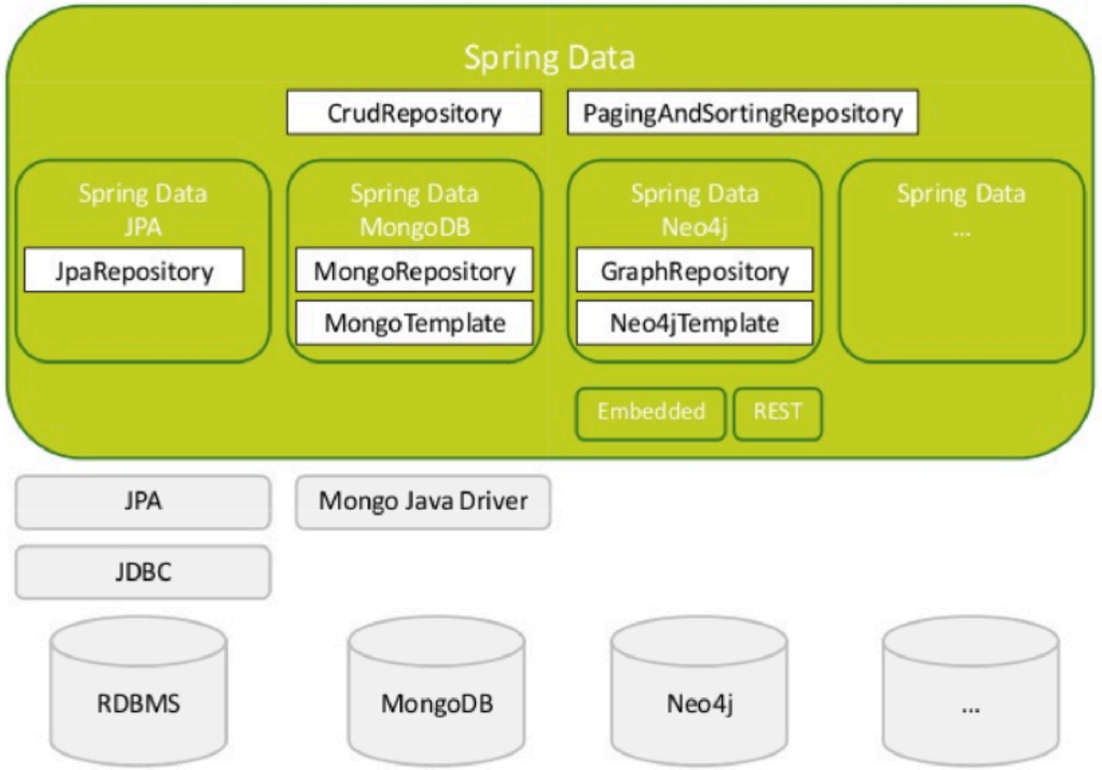
和ORM一样的概念，但提供标准化的API和查询语言（JPQL）
JPA = API + JPQL(Java Persistence Query Language) + ORM(object/relational metadata)

实现：Hibernate，Toplink，EclipseLink，Apache OpenJPA，DataNuCleus，ObjectDB

Spring Data

Spring Data旨在提供统一的，易用的数据库访问技术

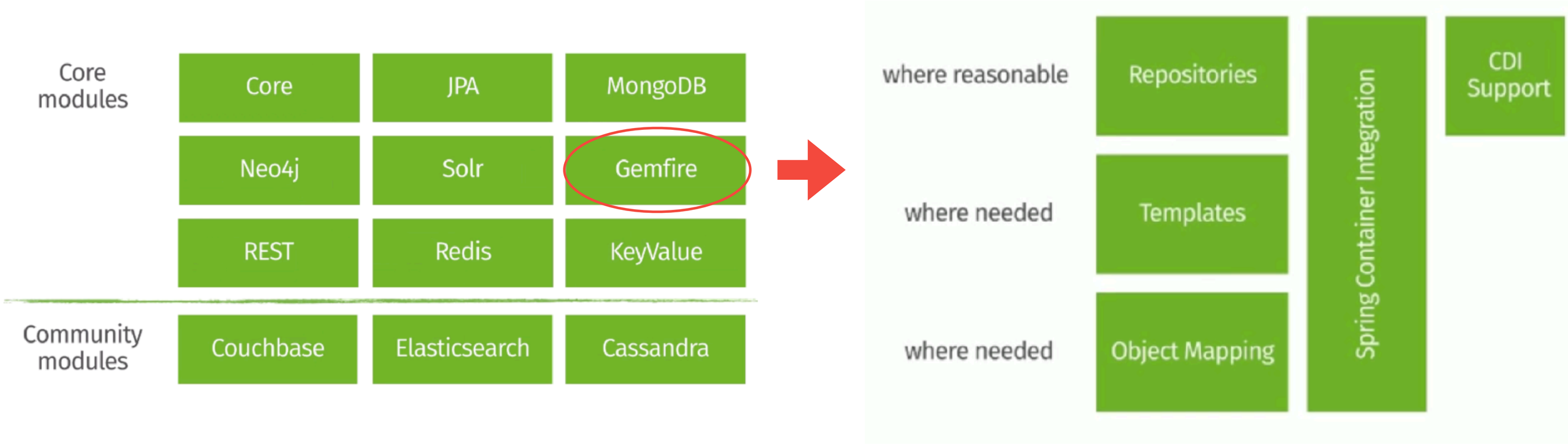




Spring Data

Spring Data架构是高度模块化的，每个核心模块有独立的贡献者

- Object mapping：将POJO转换为相应的数据格式（MongoDB的document，Neo4j的graph，Redis的property）
- Template：资源管理和异常转换（JDBCTemplate）
- Repository：提供统一的查询方式（CRUD）



Spring Data JPA

Spring data JPA是Spring data家族成员。Spring data提供了多种数据访问方式，包括关系型数据（mysql，postgres），非关系型数据库（mongodb），map-reduce，cloud service。

Spring Data JPA适合于快速创建基于JPA的用于CRUD操作的repository层，并且不用创建DAO。

Spring Data JPA特性：

1. 创建仓库repository
2. 支持QueryDSL和JPA queries
3. 领域类审计（创建和更新状态追踪）
4. 支持batch加载，排序和动态查询
5. 支持XML和Java注释两种方式
6. 通过扩展CrudRepository减少CRUD代码量

Spring Data JPA项目依赖：

1. 数据库驱动：postgresql，mysql
2. Spring框架核心：spring-core, spring-context
3. Spring REST API：spring-webmvc, jackson-databind
4. Spring数据操作：spring-data-jpa, hibernate-entitymanager

JPA vs Hibernate

Hibernate是JPA规范的一个实现，是JPA provider，而Spring Data JPA是一个JPA数据访问抽象

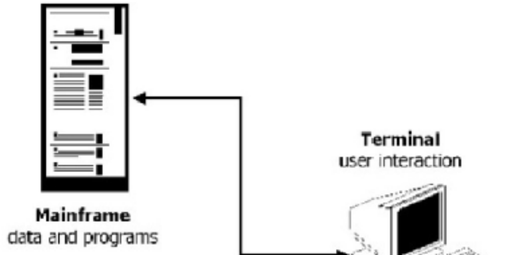
Spring Data JPA提供GenericDao的定制实现，通过方法名约定自动生成JPA查询。Spring Data JPA通过扩展repositories(crudrepository, jparepository)实现DAO接口

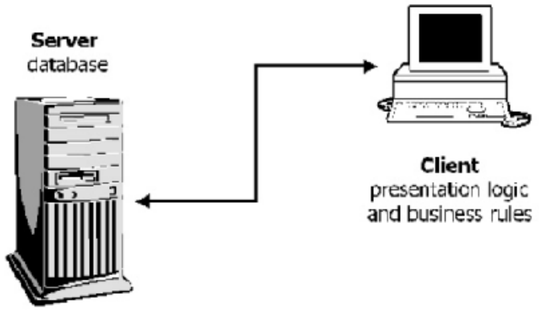
Spring JDBC是轻量级的，直接的持久化方式，目的在于实现原生SQL查询

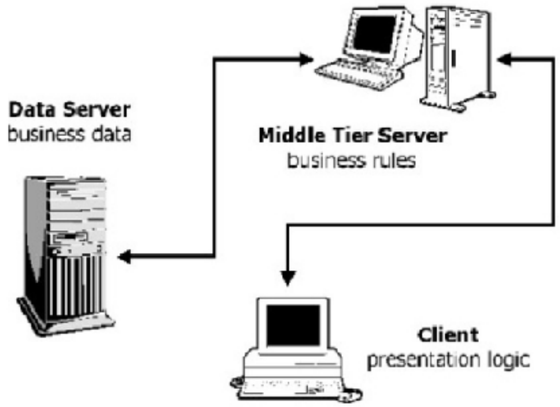
Spring Data是Spring框架的一个子项目，它的目标是简化和不同类型数据库之间的操作（CRUD），包括关系型和非关系型。Spring Data JPA是一个支持JPA的子项目，但不能单独工作，需要和Hibernate等JPA实现提供商结合才能运行

Spring Data JPA和Hibernate是互补的，而不是竞争的

分层架构
vs
MVC







Presentation layer

Spring MVC
front-springmvc

Service layer

Service layer
service-springdatajpa

Persistence layer

Spring Data JPA
persistence-springdatajpa

Database

Database

Spring 事务处理 3

事务的隔离级别

事务(Transaction)是为了保证数据的安全性，一致性，正确性

数据并发问题

1 **脏读**：A事务读取到B事务未提交的数据（B事务可能发生错误而回滚没有保存进数据库）

老板要给程序员涨工资，从*1.6*万/月涨到*2.0*万/月，但系统发生错误并没有改为*2.0*万/月，*1.6*万就打给程序员，程序员很失落。

2.**第一类更新丢失**：A事务和B事务读到同一行数据，B事务提交事务保存后，A事务更新异常而回滚导致B事务的更新丢失（旧数据）

3 **第二类更新丢失**：A事务和B事务读到同一行数据，B事务提交事务保存后，A事务提交更新导致B事务的更新丢失（未更新B提交）

老板要给程序员涨工资，从*1.6*万/月涨到*2.0*万/月，*A*事务修改纳税，*B*事务修改工资，*AB*都读到*1.6*万，*B*修改成功，但*A*仍以*1.6*万为基数计算税收

4 **不可重复读**：A事务两次读取同一行数据，因为X事务修改数据，A事务得到不同结果

5 **幻读**： A事务两次读取同一行数据，因为X事务新增数据，A事务出现幻象读（结果不一致）

程序员涨工资后，决定给女朋友买个钻石戒指，价值*1.0*万。这时女朋友正在查账，之前流水一共*2000*元，当她打印流水的时候，钻石交易成功了（新增一笔流水），所以打印出来的结果是*1.2*万，女友惊呆。

事务隔离级别

1. **Read Uncommited 可读未提交**：一个事务可以读取另一个事务未提交的数据

   程序员准备刷信用卡买书（余额1万），同时女友也刷该信用卡买化妆品（花费5000）但未提交，程序员得到结果是5000 —脏读

2. **Read Commited 可读已提交**：一个事务要等另一个事务提交后才能读取数据

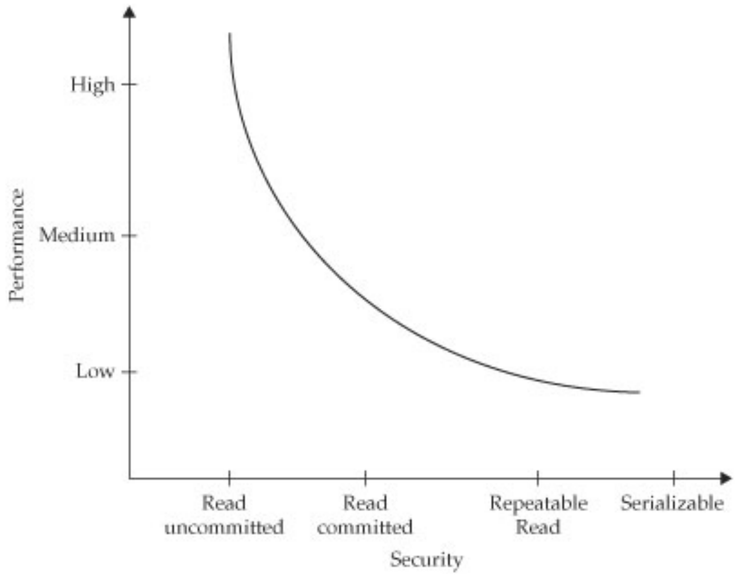
   程序员再次刷信用卡买书，买单时先检测余额（1万），同时女友买化妆品花完1万元，这时书店扣款检测时发现没钱了 — 不可重复读

3. **Repeatable Read 可重复读**：在事务执行期间会锁定该事务以任何方式引用的所有行

   女友无法买化妆品，准备查账，流水一共2000元，这时程序员购书成功消费10000元，增加一条消费记录，当女友打印消费记录时发现流水变成12000 - 幻读

4. **Serializable 串行化**：事务顺序执行，事务隔离级别最高，性能很低，一般很少使用

事务的隔离级别

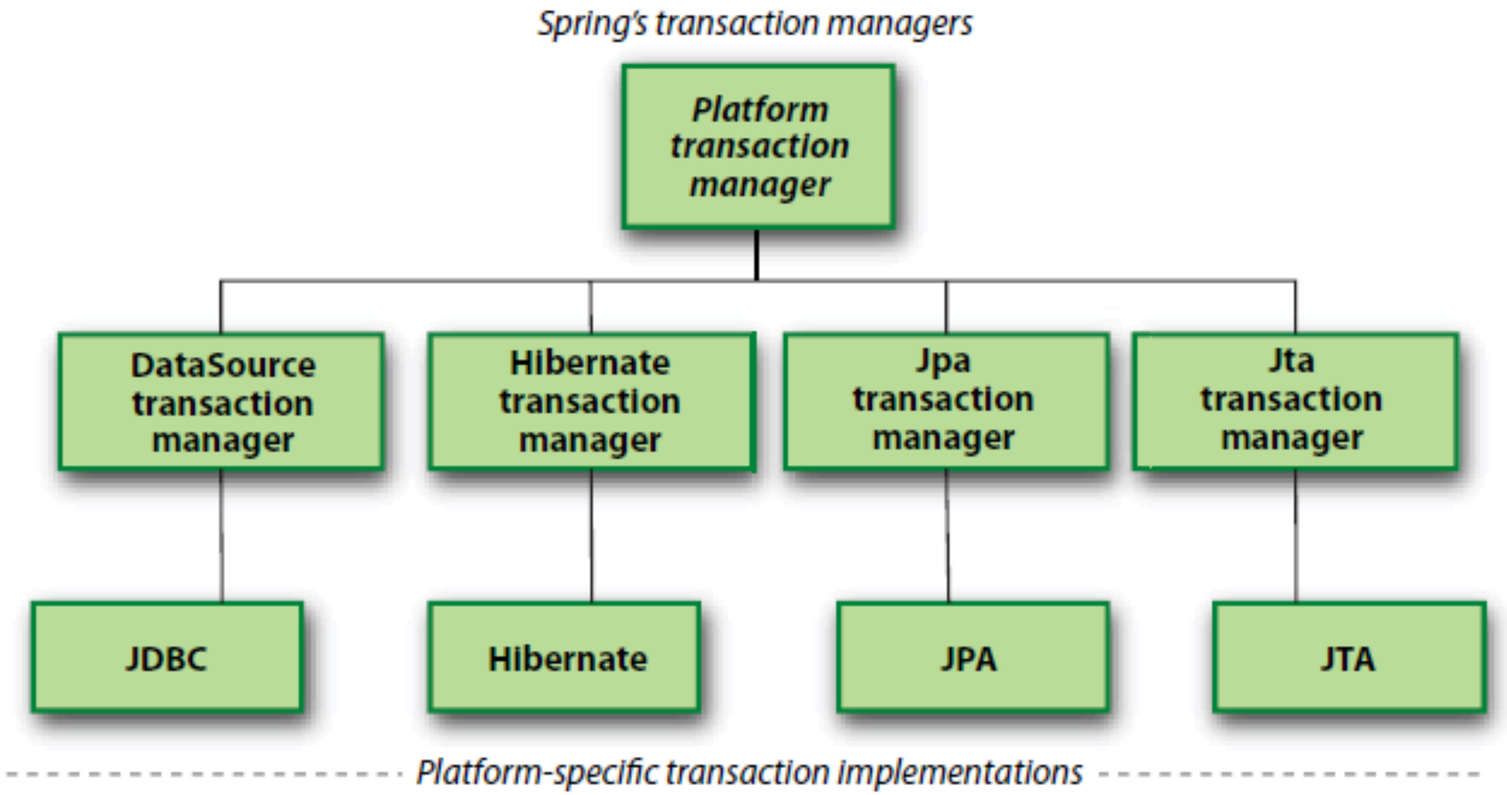


隔离级别	第一类丢失更新	脏读	可重复读	第二类丢失更新	幻读
读未提交	否	是	是	是	是
读可提交	否	否	是	是	是
可重复读	否	否	否	否	是
序列化	否	否	否	否	否

各数据库厂商的事务隔离级别定义

- Postgres默认为读可提交，理论支持四种，实际支持三种（读可提交，可重复读和序列化），选择读未提交会自动转为读可提交
- Mysql默认为可重复读，通过锁机制支持四种隔离级别，且可以修改默认隔离
- SqlServer默认为读可提交，除以上四种，微软还支持snapshot隔离（读一致性）
- Oracle默认为读可提交（一般很少修改），显示的只支持读可提交和序列化两种，还支持read only隔离（等同序列化，但只读）

Spring 事务管理



- Spring 不直接管理事务，而是提供多个事务管理器，将事务支持委托给Hibernate，JPA等持久化机制去实现特定平台的事物管理（e.g. 通过JDBC调用MySQL的事务管理）
- 声明式的事务管理，Spring使用AOP切面事务方法实现数据集成
- 可编程事务管理，实现TransactionTemplate或PlatformTransactionManager实现
- 支持大部分事务API，例如JDBC，HIbernate，JPA，JDO和JTA等，只需要选择相应的事务管理实现类，例如org.springframework.jdbc.datasource.DriverManagerDataSource

Spring 事务实现

1. Spring-tx提供spring事务管理依赖

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>${spring-framework.version}</version>
</dependency>
```

2. tx:annotation-driven通知Spring context使用基于注释的事务管理，transaction-manager定义事务管理器，默认值是transactionManager

```
<tx:annotation-driven proxy-target-class=“true" transaction-manager="transactionManager" />

<bean id="transactionManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```

3. @Transactional用于声明事务管理，可以放在类或方法上

```
public class CustomerManagerImpl implements CustomerManager {
  ...
  @Override
  @Transactional
  public void createCustomer(Customer cust) {
    customerDAO.create(cust);
  }
}
```

Spring Data JPA应用 4

PART II

Spring Data JPA

1.选择数据库，创建数据库/表，并灌入数据

2.创建资源文件database.properties，包含数据库驱动，URI，用户名/密码，ORM配置

3.创建类文件，作为spring data jpa和hibernate持久化的配置文件

4.修改Model领域模型类成为Entity，映射到对应的表

5.定义repository接口，实现CRUD，或更复杂的查询操作

6.更新service服务类调用repository接口，实现领域业务逻辑

7.更新controller调用service类，实现数据格式转换，处理和客户端的交互

数据源配置

数据库类型（驱动），数据库URI，用户名和秘密

driver=org.postgresql.Driver
url=jdbc:postgresql://127.0.0.1:5432/dharma_mall_app
user=tester
password=

hibernate持久化配置

hibernate.dialect=org.hibernate.dialect.PostgreSQL82Dialect
hibernate.show_sql=true

JPA配置

持久化配置

启动spring事务处理，指定repository位置，配置数据库连接属性文件（会放入环境变量）

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories("com.dharma.spring.repository")
@PropertySource("classpath:database.properties")
```

实体类bean工厂

配置数据源，持久化提供商（hibernate），实体类（模型类）位置

```
@Bean
LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean lfb = new LocalContainerEntityManagerFactoryBean();
    lfb.setDataSource(dataSource());
    lfb.setPersistenceProviderClass(HibernatePersistenceProvider.class);
    lfb.setPackagesToScan("com.dharma.spring.model");
    lfb.setJpaProperties(hibernateProps());
    return lfb;
}
```

JPA配置

根据数据源驱动，URI，用户名和密码配置数据源

```
@Bean
DataSource dataSource() {
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setUrl(environment.getProperty(PROPERTY_URL));
    ds.setUsername(environment.getProperty(PROPERTY_USERNAME));
    ds.setPassword(environment.getProperty(PROPERTY_PASSWORD));
    ds.setDriverClassName(environment.getProperty(PROPERTY_DRIVER));
    return ds;
}
```

配置hibernate属性

```
Properties hibernateProps() {
    Properties properties = new Properties();
    properties.setProperty(PROPERTY_DIALECT, environment.getProperty(PROPERTY_DIALECT));
    properties.setProperty(PROPERTY_SHOW_SQL, environment.getProperty(PROPERTY_SHOW_SQL));
    return properties;
}
```

领域模型

Entity是轻量级Java类，它的属性对应关系数据库一张表的字段，每个entity对象对应表的一行数据。entity之间的关系由ORM实现，可以定义在类中，也可以定义在XML文件中

JPQL是执行在entity上的查询语言，比直接写SQL简单

```
SELECT u.id, u.name FROM User u WHERE u.age > 10 AND u.age < 20
```

```
@Entity
@Table(name = "product")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "price")
    private Double price;
    @Column(name = "name")
    private String name;

    ……
}
```

定义repository接口，通过继承CrudRepository实现CRUD操作

```
public interface ProductRepository<P> extends CrudRepository<Product, Long> {
    List<Product> findByName(String name);
}
```

定义服务类@Service，调用repository实现CRUD操作和业务逻辑

```
@Service
public class ProductService {
    @Autowired
    ProductRepository<Product> productRepository;
        @Transactional
        public List<Product> getAllProducts() {
            return (List<Product>) productRepository.findAll();
        }
```

……

定义控制器@RestController，调用service得到业务处理结果，并处理用户请求和响应

```
@RestController
public class ProductController {
    @Autowired
    ProductService productService;
        @RequestMapping(value = "/product/{id}", method = RequestMethod.GET)
        public @ResponseBody Product getProduct(@PathVariable Long id) {
            return productService.getById(id);
     }
```

……

Thanks!

Any questions?