**Java学习**

**设计模式**

1. 代理模式（代码见test下的proxy包）

代理模式是常用的[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/fighterandknight/article/details/_blank)设计模式，他的特征是**代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等**。主要用来做方法的增强，让你可以在不修改源码的情况下，增强一些方法，在方法执行前后做任何你想做的事情。

1. 静态代理

在程序运行前，代理类的.class文件就已经存在了。

*/\*\*  
 \* 账户查询接口  
 \** ***@author*** *wdongsen@linesum.com  
 \** ***@data*** *2017-08-31 16:35  
 \*/*public interface ICount {  
  
 // 查询账户  
 void queryCount();  
}

*/\*\*  
 \* 账户查询接口实现(委托类)  
 \** ***@author*** *wdongsen@linesum.com  
 \** ***@data*** *2017-08-31 16:42  
 \*/*public class CountImpl implements ICount{  
  
 @Override  
 public void queryCount() {  
 System.*out*.println("");  
 }  
}

*/\*\*  
 \* 静态代理  
 \** ***@author*** *wdongsen@linesum.com  
 \** ***@data*** *2017-08-31 16:57  
 \*/*public class CountProxy implements ICount{  
  
 private CountImpl countImpl;  
  
 public CountProxy(CountImpl countImpl) {  
 this.countImpl = countImpl;  
 }  
  
 @Override  
 public void queryCount() {  
 System.*out*.println("处理之前。。。");  
 countImpl.queryCount();  
 System.*out*.println("处理之后。。。");  
 }  
}

**总结:**  
1.可以做到在不修改目标对象的功能前提下,对目标功能扩展.  
2.缺点:因为**代理对象需要与目标对象实现一样的接口**，所以会有很**多代理类**。同时，一旦**接口增加方法**，目标对象与代理对象都要维护。

1. jdk动态代理

代理类所在包:java.lang.reflect.Proxy  
JDK实现代理只需要使用newProxyInstance方法,但是该方法需要接收三个参数,完整的写法是:

static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,InvocationHandler h )

注意该方法是在Proxy类中是静态方法,且接收的三个参数依次为:

ClassLoader loader,:指定当前目标对象使用类加载器,获取加载器的方法是固定的

Class<?>[] interfaces,:目标对象实现的接口的类型,使用泛型方式确认类型

InvocationHandler h:事件处理,执行目标对象的方法时,会触发事件处理器的方法,会把当前执行目标对象的方法作为参数传入

*/\*\*  
 \* 创建动态代理对象（也可以实现InvocationHandler接口）  
 \** ***@author*** *wdongsen@linesum.com  
 \** ***@data*** *2017-09-07 9:27  
 \*/*public class DynamicProxyFactory{  
  
 // 目标对象  
 private Object target;  
  
 public DynamicProxyFactory(Object target) {  
 this.target = target;  
 }  
  
 // 生成代理对象  
 public Object getProxyInstance() {  
 return Proxy.*newProxyInstance*(target.getClass().getClassLoader(),  
 target.getClass().getInterfaces(),  
 (proxy, method, args) -> {  
 System.*out*.println("before...");  
 // 执行目标方法  
 Object result = method.invoke(target, args);  
 System.*out*.println("after...");  
 return result;  
 });  
 }  
}

**总结:**  
1.代理对象,**不需要实现接口**2.代理对象的生成,是利用JDK的API,动态的在内存中构建代理对象(需要我们指定创建代理对象/目标对象实现的接口的类型)  
3.动态代理也叫做:JDK代理,接口代理

1. cglib动态代理

静态代理和jdk动态代理模式都是要求目标对象是**实现一个接口的目标对象**,但是有时候目标对象只是一个单独的对象,并没有实现任何的接口,这个时候就可以使用以目标对象子类的方式类实现代理,这种方法就叫做:Cglib代理。

Cglib是一个强大的高性能的代码生成包,它可以在**运行期扩展java类与实现java接口**.它广泛的被许多AOP的框架使用,例如Spring AOP和synaop,为他们提供方法的interception(拦截)。Cglib包的底层是通过使用一个小而块的字节码处理框架**ASM**来转换字节码并生成新的类.不鼓励直接使用ASM,因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉.

tip:1、代理的类不能为final,否则报错。2、目标对象的方法如果为final/static,那么就不会被拦截,即不会执行目标对象额外的业务方法.

*/\*\*  
 \* cglib动态代理工厂  
 \** ***@author*** *wdongsen@linesum.com  
 \** ***@data*** *2017-09-07 10:03  
 \*/*public class CglibProxyFactory implements MethodInterceptor {  
  
 // 目标对象  
 private Object object;  
  
 public CglibProxyFactory(Object object) {  
 this.object = object;  
 }  
  
 */\*\*  
 \* 给目标对象创建一个代理对象  
 \** ***@return*** *\*/* public Object getProxyInstance(){  
 // 工具类  
 Enhancer enhancer = new Enhancer();  
 // 设置父类  
 enhancer.setSuperclass(object.getClass());  
 // 设置回调函数  
 enhancer.setCallback(this);  
 // 创建子类（代理对象）  
 return enhancer.create();  
 }  
  
  
 @Override  
 public Object intercept(Object o, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {  
 System.*out*.println("before......");  
 Object result = method.invoke(object, args);  
 System.*out*.println("after.....");  
 return result;  
 }  
}

在Spring的AOP编程中:  
如果加入容器的目标对象有实现接口,用JDK代理  
如果目标对象没有实现接口,用Cglib代理

**Mysql数据库**

1. 数据库引擎

（参考：http://www.cnblogs.com/0201zcr/p/5296843.html）

查看当前数据库支持的引擎和默认的数据库引擎：show engines;

* 1. Innodb

Innodb引擎提供了对数据库**ACID事务**的支持，并且实现了SQL标准的四种隔离级别。该引擎还提供了**行级锁**和外键约束，不支持FULLTEXT类型的索引，而且它没有保存表的行数，当**SELECT COUNT(\*) FROM TABLE时需要扫描全表**。  
 为了更好地理解ACID，下面以银行账户转账为例进行说明：

-- 开始事务

START TRANSACTION;

-- 查询支票账户余额

SELECT balance FROM checking WHERE customer\_id = 10233276;

-- 将支票账户减去200

UPDATE checking SET balance = balance - 200.00 WHERE customer\_id = 10233276;

-- 将余额账户增加200

UPDATE savings SET balance = balance + 200.00 WHERE customer\_id = 10233276;

-- 提交事务更新COMMIT;

**原子性**：事务中的操作是一个不可分割的整体单元，要么全部都做，要么全部不做。要么完全提交（10233276的checking余额减少200，savings 的余额增加200），要么完全回滚（两个表的余额都不发生变化）。

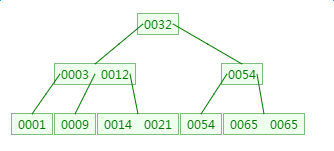
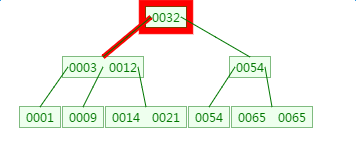
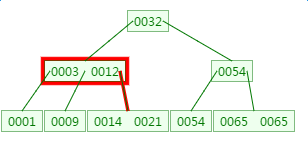
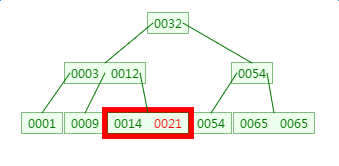
**一致性**：事务执行前后数据库都必须处于一致性状态。这个例子的一致性体现在 200元不会因为数据库系统运行到第3行之后，第4行之前时崩溃而不翼而飞，因为事物还没有提交。

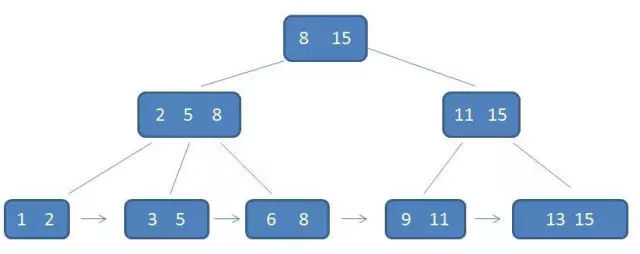
**隔离性**：一个事物所做的修改在最终提交之前对其余事务是不可见的。允许在一个事务中的操作语句会与其他事务的语句隔离开，比如事务A运行到第3行之后，第4行之前，此时事务B去查询checking余额时，它仍然能够看到在事务A中被减去的200元（账户钱不变），因为事务A和B是彼此隔离的。在事务A提交之前，事务B观察不到数据的改变。

**持久性**：一旦事务提交完成，修改就是永久的，即使服务器宕机也不会影响到。

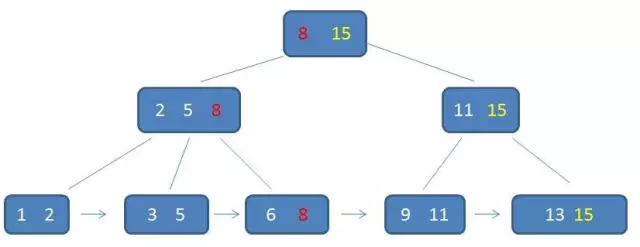
* 1. MyIsam

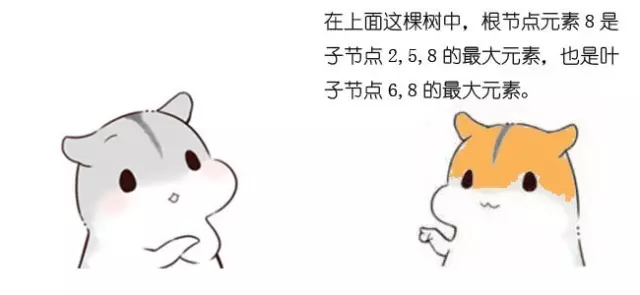
没有提供对数据库事务的支持，也**不支持行级锁和外键**，因此当**INSERT(插入)或UPDATE(更新)数据时即写操作需要锁定整个表**，效率便会低一些。不过和Innodb不同，MyIASM中存储了表的行数，于是SELECT COUNT(\*) FROM TABLE时只需要直接读取已经保存好的值而不需要进行全表扫描。如果表的读操作远远多于写操作且不需要数据库事务的支持，那么MyIASM也是很好的选择。

1. 索引
   1. B-Tree（多路搜索树）  
      B-Tree就是我们常说的B树，这种数据结构常常用于实现数据库索引，因为它的查找效率比较高。
      1. 磁盘IO与预读   
         磁盘读取依靠的是机械运动，分为**寻道时间、旋转延迟、传输时间**三个部分，这三个部分耗时相加就是一次磁盘IO的时间，大概9ms左右。这个成本是访问内存的十万倍左右；正是由于磁盘IO是非常昂贵的操作，所以计算机操作系统对此做了优化：**预读**；每一次IO时，不仅仅把当前磁盘地址的数据加载到内存，同时也把相邻数据也加载到内存缓冲区中。因为局部预读原理说明：当访问一个地址数据的时候，与其相邻的数据很快也会被访问到。每次磁盘IO读取的数据我们称之为一页（page）。一页的大小与操作系统有关，一般为4k或者8k。这也就意味着读取一页内数据的时候，实际上发生了一次磁盘IO。
      2. B-Tree与二叉查找树的对比  
         我们知道二叉查找树查询的时间复杂度是O（logN），查找速度最快和比较次数最少，既然性能已经如此优秀，但为什么实现索引是使用B-Tree而不是二叉查找树，关键因素是磁盘IO的次数。**数据库索引是存储在磁盘上，当表中的数据量比较大时，索引的大小也跟着增长，达到几个G甚至更多。当我们利用索引进行查询的时候，不可能把索引全部加载到内存中，只能逐一加载每个磁盘页，这里的磁盘页就对应索引树的节点。减少磁盘IO的次数就必须要压缩树的高度，让瘦高的树尽量变成矮胖的树，所以B-Tree就在这样伟大的时代背景下诞生了。**
      3. B-Tree   
         m阶B-Tree满足以下条件：  
         1、**每个节点最多拥有m个子树** 2、根节点至少有2个子树  
         3、**分支节点至少拥有m/2颗子树**（除根节点和叶子节点外都是分支节点）  
         4、**所有叶子节点都在同一层、每个节点最多可以有m-1个key**，并且以升序排列  
          如下有一个3阶的B树，观察查找元素21的过程：  
           
         第一次磁盘IO：     
           
         第二次磁盘IO：  
           
         这里有一次内存比对：分别跟3与12比对  
         第三次磁盘IO:  
           
         比对是在内存中完成中，不涉及到磁盘IO，耗时可以忽略不计。另外B树种一个节点中可以存放很多的key（个数由树阶决定）。相同数量的key在B树中生成的节点要远远少于二叉树中的节点，相差的节点数量就等同于磁盘IO的次数。这样到达一定数量后，性能的差异就显现出来了。
   2. B+Tree
      1. 特征  
         1.有k个子树的中间节点**包含有k个元素（B树中是k-1个元素），每个元素不保存数据，只用来索引，所有数据都保存在叶子节点**。  
         2.所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。  
         3.**所有的中间节点元素都同时存在于子节点，在子节点元素中是最大（或最小）元素**。
      2. 优势  
         1.单一节点存储更多的元素，使得查询的IO次数更少。  
         2.所有查询都要查找到叶子节点，查询性能稳定。  
         3.所有叶子节点形成有序链表，便于范围查询。
      3. 结构

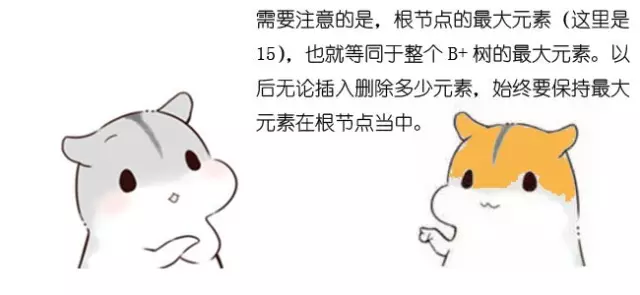


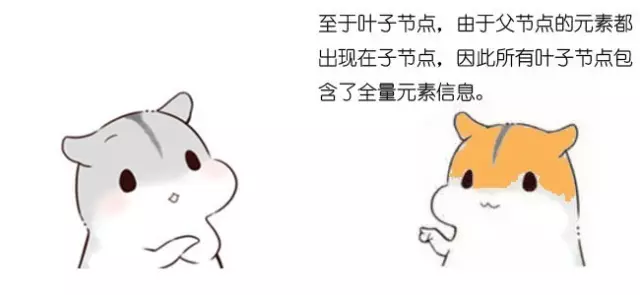




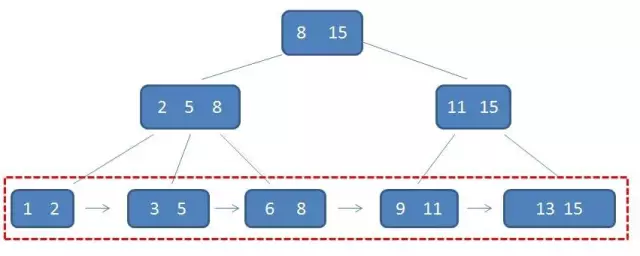






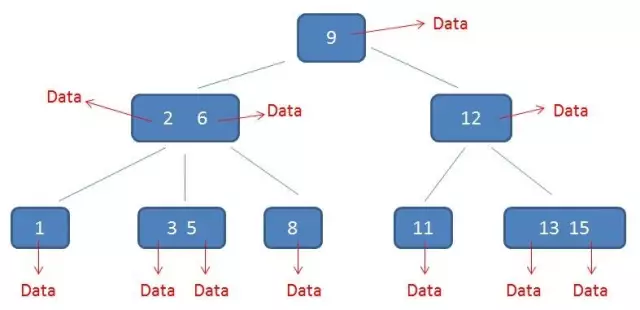


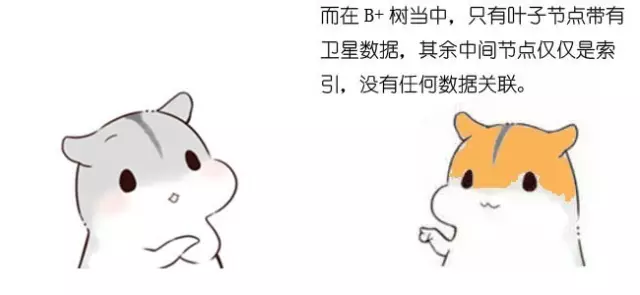
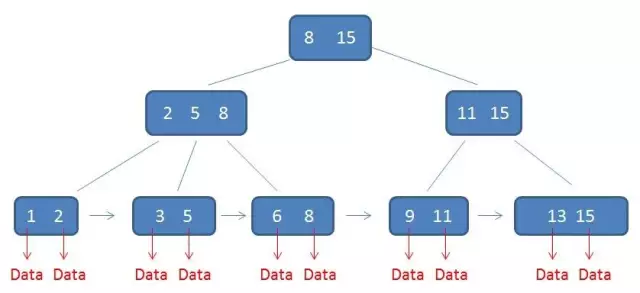








B-树中的卫星数据（Satellite Information）：

B+树中的卫星数据（Satellite Information）：

需要补充的是，在数据库的聚集索引（Clustered Index）中，叶子节点直接包含卫星数据。在非聚集索引（NonClustered Index）中，叶子节点带有指向卫星数据的指针。

1. 索引