# 一、先定义一个cache标准接口

*//定义一个缓存标准对象*public interface Cache {  
  
 int size();  
  
 void putObject(Object key,Object val);  
  
 Object getObject(Object key);  
  
 Object removeObject(Object key);  
  
 void clear();  
}

# 二、设计基础存储功能的PertetualCache

*/\*  
缓存对象：  
 1）存储结构  
 2）淘汰算法：无（直到内存溢出）  
 \*/*public final class PertetualCache implements Cache{  
 private HashMap<Object,Object> cache = new HashMap<>();  
 @Override  
 public int size() {  
 return cache.size();  
 }  
 @Override  
 public void putObject(Object key, Object val) {  
 cache.put(key,val);  
 }  
 @Override  
 public Object getObject(Object key) {  
 return cache.get(key);  
 }  
 @Override  
 public Object removeObject(Object key) {  
 return cache.remove(key);  
 }  
 @Override  
 public void clear() {  
 cache.clear();  
 }  
  
 @Override  
 public String toString() {  
 return cache.toString();  
 }  
}

# 三、设计日志功能的LogCache

*/\*此cache对象记录命中率（访问）\*/*public class LogCache implements Cache{  
 private int requests;  
 private int hits;  
 private Cache cache;  
 public LogCache(Cache cache){  
 this.cache = cache;  
 }  
 @Override  
 public int size() {  
 return cache.size();  
 }  
 @Override  
 public void putObject(Object key, Object val) {  
 cache.putObject(key,val);  
 }  
 @Override  
 public Object getObject(Object key) {  
 *//每一次请求都记录* requests++;  
 Object obj = cache.getObject(key);  
 if(obj != null){  
 *//记录成功次数* hits++;  
 }  
 System.*out*.println("命中率："+1.0\*hits/requests);  
 return obj;  
 }  
 @Override  
 public Object removeObject(Object key) {  
 return cache.removeObject(key);  
 }  
 @Override  
 public void clear() {  
 cache.clear();  
 }  
 @Override  
 public String toString() {  
 return cache.toString();  
 }  
}  
class TestLogCache{  
 public static void main(String[] args) {  
 Cache cache = new LogCache(new PertetualCache());*//装饰者模式* cache.putObject("a",1);  
 cache.putObject("b",1);  
 cache.putObject("c",1);  
 cache.getObject("d");*//命中率：0.0* cache.getObject("a");*//命中率:0.5* }  
}

# 四、设计线程安全的SynchronizedCache

public class SynchronizedCache implements Cache{  
 private Cache cache;  
 public SynchronizedCache(Cache cache){  
 this.cache = cache;  
 }  
 @Override  
 public synchronized int size() {  
 return cache.size();  
 }  
 @Override  
 public synchronized void putObject(Object key, Object val) {  
 cache.putObject(key,val);  
 }  
 @Override  
 public synchronized Object getObject(Object key) {  
 return cache.getObject(key);  
 }  
 @Override  
 public synchronized Object removeObject(Object key) {  
 return cache.removeObject(key);  
 }  
 @Override  
 public synchronized void clear() {  
 cache.clear();  
 }  
 @Override  
 public String toString() {  
 return cache.toString();  
 }  
}  
class TestSynchronizedCache{  
 public static void main(String[] args) {  
 Cache cache = new SynchronizedCache(new FifiCache(new LogCache(new PertetualCache()),3));  
 cache.putObject("a",1);  
 cache.putObject("b",1);  
 cache.putObject("c",1);  
 cache.getObject("b");  
 cache.getObject("d");  
 cache.putObject("d",1);  
 System.*out*.println(cache);  
 }  
}

# 五、设计先进先出算法的FifoCache

public class FifiCache implements Cache{  
 private int maxCap;  
 private Cache cache;  
 private Deque<Object> keyOrder = new LinkedList<>();  
 *//设计构造器* public FifiCache(Cache cache,int maxCap){  
 this.cache = cache;  
 this.maxCap = maxCap;  
 }  
 @Override  
 public int size() {  
 return cache.size();  
 }  
 @Override  
 public void putObject(Object key, Object val) {  
 *//记录key* keyOrder.addLast(key);  
 *//判断是否满* if(size()>=maxCap){  
 *//最老的key* Object oldestKey = keyOrder.removeFirst();  
 *//删除最老元素* cache.removeObject(oldestKey);  
 }  
 cache.putObject(key,val);  
 }  
 @Override  
 public Object getObject(Object key) {  
 return cache.getObject(key);  
 }  
 @Override  
 public Object removeObject(Object key) {  
 keyOrder.remove(key);  
 return cache.removeObject(key);  
 }  
 @Override  
 public void clear() {  
 keyOrder.clear();  
 cache.clear();  
 }  
 @Override  
 public String toString() {  
 return cache.toString();  
 }  
}  
class TestFifiCache{  
 public static void main(String[] args) {  
 Cache cache = new FifiCache(new LogCache(new PertetualCache()),3);  
 cache.putObject("a",1);  
 cache.putObject("b",1);  
 cache.putObject("c",1);  
 cache.getObject("b");  
 cache.getObject("d");  
 cache.putObject("d",1);  
 System.*out*.println(cache);  
 }  
}

# 六、设计Lru算法的LruCache

public class LruCache implements Cache{  
 private int maxCap;  
 private Cache cache;  
 *//保存最老的key* private Object oldestKey;  
 *//记录访问顺序* private Map<Object,Object> keyMap;  
 *//构造函数* public LruCache(Cache cache,int maxCap){  
 this.maxCap = maxCap;  
 this.cache = cache;  
 *//记录访问顺序true* keyMap = new LinkedHashMap<>(maxCap,0.75f,true){  
 @Override  
 protected boolean removeEldestEntry(Map.Entry<Object, Object> eldest) {  
 boolean tooBig = size()>maxCap;  
 if (tooBig){  
 *//获取最老key* oldestKey = eldest.getKey();  
 }  
 return tooBig;  
 }  
 };  
 }  
 @Override  
 public int size() {  
 return cache.size();  
 }  
 @Override  
 public void putObject(Object key, Object val) {  
 *//添加数据* cache.putObject(key,val);  
 *//记录key* keyMap.put(key,key);*//这步已经完成最老key的记录  
 //判断是否有最老key* if(oldestKey != null){  
 *//删除* cache.removeObject(oldestKey);  
 }  
 }  
 @Override  
 public Object getObject(Object key) {  
 *//记录访问,这步已完成访问记录* Object cacheKey = keyMap.get(key);  
 return cache.getObject(cacheKey);  
 }  
 @Override  
 public Object removeObject(Object key) {  
 keyMap.remove(key);  
 return cache.removeObject(key);  
 }  
 @Override  
 public void clear() {  
 keyMap.clear();  
 cache.clear();  
 }  
 @Override  
 public String toString() {  
 return cache.toString();  
 }  
}  
class TestLruCache{  
 public static void main(String[] args) {  
 Cache cache = new SynchronizedCache(new LruCache(new LogCache(new PertetualCache()),3));  
 cache.putObject("a",1);  
 cache.putObject("b",1);  
 cache.putObject("c",1);  
 cache.getObject("b");*//acb* cache.getObject("d");  
 cache.putObject("d",1);*//cbd  
 //这里的输出顺序与预料不同，因为存数据的cache没有排序，访问顺序在keyMap中体现* System.*out*.println(cache);  
 }  
}