public class ObjectPool<T extends ObjectPool.Poolable> {

// 下一个ObjectPool 的 id

private static int ids = 0;

// 当前 ObjectPool 的id

private int poolId;

private int desiredCapacity;

private Object[] objects;

private int objectsPointer;

private T modelObject;

private float replenishPercentage;

/\*\*

\* Returns the id of the given pool instance.

\*

\* @return an integer ID belonging to this pool instance.

\*/

public int getPoolId(){

return poolId;

}

/\*\*

\* Returns an ObjectPool instance, of a given starting capacity, that recycles instances of a given Poolable object.

\*

\* @param withCapacity A positive integer value.

\* @param object An instance of the object that the pool should recycle.

\* @return

\*/

public static synchronized ObjectPool create(int withCapacity, Poolable object){

ObjectPool result = new ObjectPool(withCapacity, object);

result.poolId = ids;

// 创建过一个 ObjectPool 后，ids自加

ids++;

return result;

}

// capacity 大小和 Poolable 的实体类，权限为private ，因此实例化该类只能用上面那 // 个方法

private ObjectPool(int withCapacity, T object){

if(withCapacity <= 0){

throw new IllegalArgumentException("Object Pool must be instantiated with a capacity greater than 0!");

}

this.desiredCapacity = withCapacity;

this.objects = new Object[this.desiredCapacity];

this.objectsPointer = 0;

this.modelObject = object;

this.replenishPercentage = 1.0f;

this.refillPool();

}

/\*\*

\* Set the percentage of the pool to replenish on empty. Valid values are between

\* 0.00f and 1.00f

\*

\* @param percentage a value between 0 and 1, representing the percentage of the pool to replenish.

\*/

public void setReplenishPercentage(float percentage){

float p = percentage;

if(p > 1){

p = 1;

}

else if(p < 0f){

p = 0f;

}

this.replenishPercentage = p;

}

public float getReplenishPercentage(){

return replenishPercentage;

}

private void refillPool(){

this.refillPool(this.replenishPercentage);

}

// 填充数值的百分比

private void refillPool(float percentage){

int portionOfCapacity = (int) (desiredCapacity \* percentage);

if(portionOfCapacity < 1){

portionOfCapacity = 1;

}else if(portionOfCapacity > desiredCapacity){

portionOfCapacity = desiredCapacity;

}

for(int i = 0 ; i < portionOfCapacity ; i++){

this.objects[i] = modelObject.instantiate();

}

objectsPointer = portionOfCapacity - 1;

}

/\*\*

\* Returns an instance of Poolable. If get() is called with an empty pool, the pool will be

\* replenished. If the pool capacity is sufficiently large, this could come at a performance

\* cost.

\*

\* @return An instance of Poolable object T

\*/

public synchronized T get(){

if(this.objectsPointer == -1 && this.replenishPercentage > 0.0f){

this.refillPool();

}

T result = (T)objects[this.objectsPointer];

result.currentOwnerId = Poolable.NO\_OWNER;

this.objectsPointer--;

return result;

}

/\*\*

\* Recycle an instance of Poolable that this pool is capable of generating.

\* The T instance passed must not already exist inside this or any other ObjectPool instance.

\*

\* @param object An object of type T to recycle

\*/

public synchronized void recycle(T object){

if(object.currentOwnerId != Poolable.NO\_OWNER){ // 如果object 属于Poolable

if(object.currentOwnerId == this.poolId){

throw new IllegalArgumentException("The object passed is already stored in this pool!");

}else {

throw new IllegalArgumentException("The object to recycle already belongs to poolId " + object.currentOwnerId + ". Object cannot belong to two different pool instances simultaneously!");

}

}

this.objectsPointer++;

if(this.objectsPointer >= objects.length){

this.resizePool();

}

object.currentOwnerId = this.poolId;

objects[this.objectsPointer] = object;

}

/\*\*

\* Recycle a List of Poolables that this pool is capable of generating.

\* The T instances passed must not already exist inside this or any other ObjectPool instance.

\*

\* @param objects A list of objects of type T to recycle

\*/

public synchronized void recycle(List<T> objects){

while(objects.size() + this.objectsPointer + 1 > this.desiredCapacity){

this.resizePool();

}

final int objectsListSize = objects.size();

// Not relying on recycle(T object) because this is more performant.

for(int i = 0 ; i < objectsListSize ; i++){

T object = objects.get(i);

if(object.currentOwnerId != Poolable.NO\_OWNER){

if(object.currentOwnerId == this.poolId){

throw new IllegalArgumentException("The object passed is already stored in this pool!");

}else {

throw new IllegalArgumentException("The object to recycle already belongs to poolId " + object.currentOwnerId + ". Object cannot belong to two different pool instances simultaneously!");

}

}

object.currentOwnerId = this.poolId;

this.objects[this.objectsPointer + 1 + i] = object;

}

this.objectsPointer += objectsListSize;

}

private void resizePool() {

final int oldCapacity = this.desiredCapacity;

this.desiredCapacity \*= 2;

Object[] temp = new Object[this.desiredCapacity];

for(int i = 0 ; i < oldCapacity ; i++){

temp[i] = this.objects[i];

}

this.objects = temp;

}

/\*\*

\* Returns the capacity of this object pool. Note : The pool will automatically resize

\* to contain additional objects if the user tries to add more objects than the pool's

\* capacity allows, but this comes at a performance cost.

\*

\* @return The capacity of the pool.

\*/

public int getPoolCapacity(){

return this.objects.length;

}

/\*\*

\* Returns the number of objects remaining in the pool, for diagnostic purposes.

\*

\* @return The number of objects remaining in the pool.

\*/

public int getPoolCount(){

return this.objectsPointer + 1;

}

public static abstract class Poolable{

public static int NO\_OWNER = -1;

int currentOwnerId = NO\_OWNER;

protected abstract Poolable instantiate();

}

}