PoolRoadMap

Pool Road Map

Releases

Pool 2

This will not fully compatible with the Pool 1.x releases. This has been necessitated by some recent changes to semantics. Areas with the code contracts defined in interfaces is loosely defined or unnecessary allows methods to return a variety of responses with the same meaning will be changed to more user friendly behavior. This release will take advantage of Java 1.4 features.

The planned changes of behavior are based on the intent of the client code using the pool.

Acquiring idle objects:

- addObject: client code calling this method wants to add another object to the list of idle objects. If this fails then the client code should know about it with an exception.
- borrowObject: client code calling this method wants a valid object. Implementations should try to return an idle object first. If this idle object fails to be activated or validated then it should be destroyed and the pool should try again. When there are no more idle objects then a new object should be created. If this fails then an exception should be propagate to the client code. Also, null should never be returned.

Returning idle objects:

- returnObject: when client code is done with an object and still thinks the object is valid it passes it back to the pool with returnObject. returnObject passivates the instance and adds it to the idle object pool. If the passivation fails then the object is not added to the idle object pool and it an attempt to destroy the object is made. In no situation should an exception be allowed to propagate to client code. The client code is returning the object and doesn't care about it anymore. If the client code need information about objects that fail to be returned it should handle that in the PoolableObjectFactory implementation.
- invalidateObject: when client code is done with an object and thinks it's invalid for future reuse. The object should be destroyed and never should an exception be propagated to the calling code.

Pool maintenance methods:

- clear: like addObject the client code is requesting a specific behavior and should be notified of problems clearing the pool. This does not include exceptions from destroying objects. (That said, I personally cannot think of a situation where a clear would fail.)
- close: the client code calling this method is done with the pool. Resources that it makes sense to free should be freed. (That does not mean nulling all private fields as the factory will still be needed to destroyObjects returned to the pool after it has been closed.) Unlike the close method, problems while freeing those resources should be swallowed. After calling the close method addObject and borrowObject method should throw an IllegalStateException and the other methods like returnObject or invalidateObject should accept objects but immediately destroy and discard them. This is important in a multi-threaded enviroment, such as a webapp, where it is difficult or impossible to be sure when any object borrowed before the call to close will be returned and threads that happen to return objects after the pool has been closed should be allowed to complete normally until they can be shutdown.
- getNumActive and getNumIdle: previously these were declared to throw exceptions when they were unsupported. Instead they should return a negative value because it would never make sense for the active or idle count of a pool to go negative. By not throwing an exception code like: if (pool.getNumActive() > 3) ... is safer and less error prone.
- setFactory: I think it's a gross misuse of a pool to change it's factory after it's been created, especially after the first object has been borrowed.

Other points:

- a pool implementation should never allow a null PoolableObjectFactry. A pool with no factory is called a java.util.List or a keyed pool a java.util.Map of java.util.Lists.
- any object borrowed from the pool must be returned to the pool. Client code should be careful to only return a borrowed object once. The same borrowed instance should never be returned to both returnObject and invalidateObject in one borrow cycle.
- an object that fails to fails activation, validation, passivation, or is passed to invalidateObject must be destroyed. Much like the finalize method isn't guarenteed to be called in a deterministic manner, neither is when the call to destroyObject is to be made strictly specified. For example a pool could queue objects to be destroyed with a java.util.Timer to help improve the perceived performance of the pool.

Configurations:

- Make the Config instances immutable and only used as structs for pools ctors;
- Some subset of the config properties (need to decide) need to be *individually* mutable at runtime - e.g., setMaxActive(newMaxActive) needs to remain. We have agreed at this point that at least maxActive and maxWait need to be runtime mutable;
- Correct functioning of the pool with the current implementation requires that no thread can change maxActive while another thread holds the lock on the pool's monitor. Just making the properties volatile or protecting them with another lock will cause problems;
- remove the (pool-synchronized) reconfigure(Config) to enable multiple properties to be changed atomically;
- GOP has the following properties:
    - numActive
    - numIdle
    - maxActive
    - maxIdle
    - minIdle
    - maxWait
    - whenExhaustedAction
    - minEvictableIdleTimeMillis

    - softMinEvictableIdleTimeMillis
    - testOnBorrow
    - testOnReturn
    - testWhileIdle
    - timeBetweenEvictionRunsMillis
    - numTestsPerEvictionRun
    - lifo
    - factory
      The first two are initialized at 0 and are read-only. All of the others are read/write with default values. We have decided to deprecate the setter for the factory property and we have not decided yet which others should be non-final. Here is the suggested list of properties that should be runtime mutable:
    - maxTotal
    - maxWait
    - maxIdle
    - minIdle
    - idleTimeout
    - softIdleTimeout
      Proposal: rename maxActive to maxTotal, since the name is confusing, as it really represents the limit on the total number of object instances (idle or "active") under management by the pool. While maxActive does in fact limit numActive, they actually measure different things, so the names should be different. The others should be obvious.
- GKOP has all of the above properties, but adds
    - maxTotal
      and has parameterized properties
    - numIdle(key)
    - numActive(key)
    - maxActive, maxIdle, minIdle are binding per key.
      Proposal for GKOP
    - s/maxActive/maxTotalPerKey
    - s/maxIdle/maxIdlePerKey
    - s/minIdle/minIdlePerKey

JMX support:

- a UUID has to be assigned each pool to represent multiple pools in a JVM;
- use the UUID to determine exactly which one you're talking about, but use the name when displaying it to the user;

- A resultant ObjectName is like: domain=[optionalProvidedName|org.apache.commons.pool.poolType]:uuid=[uuidValue]

Pool 3

This may break API compatibility for implementations of pools but shouldn't affect client code using pools. Mostly the exceptions declared in the interfaces that is no longer needed because of behavior changes in Pool 2 will be removed. This release probably will take advantage of Java 5 (aka: 1.5) features.

Probable changes:

- Generification
- JMX monitoring: to be able to answer questions like: how many DB connects are currently in use?
- java.util.concurrent: improved parallelism

Performant Implementations

The current implementations concentrate on stability and robustness rather than performance. Pool 2 will introduce new implementations geared towards performance.