Nedis-Sentinel

What is nedis-sentinel

nedis-sentinel 是基于 redis2.8 的 java 客户端，具有池化功能，负载均衡，高可用性，容错处理特性，并对高并发场景的支持做了大量的优化工作。

nedis 基于 redis 的 sentinel 来实现增删改查操作，这样设计的好处来于能非常容易的实现高可用和容错的特性，随之而来的代价是，redis 必须使用在 sentinel 的情况下才能使用 nedis-sentinel。

Who use nedis-sentinel

如果你需要在项目中使用 redis，并需要进行增删改查操作，都可以使用 nedis-sentinel 帮助你减少开发上的开销。由于 nedis-sentinel 的诸多特性使得访问 redis 变得简单，高效，可靠。

Nedis-Sentinel performance

以下是对 nedis-sentinel 的 3 次性能测试结果，则是场景为循环 1000 次简单的 k-v 查询的耗时情况。

```
2.0 test
nedis2.0.0 query test
detail:query string in index 1, and key is "test1", return value is "value_test1"
count(10000) all:3366 ms ave: 0.3366
```

```
2.0 test
nedis2.0.0 query test
detail:query string in index 1, and key is "test1", return value is "value_test1"
count(10000) all:3036 ms ave: 0.3036
```

```
2.0 test
nedis2.0.0 query test
detail:query string in index 1, and key is "test1", return value is "value_test1"
count(10000) all:3323 ms ave: 0.3323
```

可以看出一次简单的 k-v 查询 < 0.34ms。

Get start

使用 nedis-sentinel 不需要复杂的配置，只要 3 步骤就可以实现。

Maven

第一步是添加 maven 依赖（source 可选）

jar：

```
<dependency>
        <groupId>com.newegg.ec</groupId>
        <artifactId>nedis-sentinel</artifactId>
        <version>2.0.8</version>
</dependency>
```

source:

```
<dependency>
   <groupId>com.newegg.ec</groupId>
   <artifactId>nedis-sentinel</artifactId>
   <version>2.0.8</version>
   <classifier>sources</classifier>
</dependency>
```

Config

使用 nedis-sentinel 的需要将一份名为 sentinel.xml 的配置文件添加的项目的 classpath 中。

sentinel.xml 的内容为大致为如下：

```
<nedis>
    <sentinels masterName="master" reflashWaitTimeMillis="5000" soTimeout="3000" slaveBackUp="false">
        <sentinel host="10.16.40.61" port="8371" />
        <sentinel host="10.16.40.54" port="8371" />
        <sentinel host="10.16.40.55" port="8371" />
        <sentinel host="10.16.40.57" port="8371" />
    </sentinels>
    <slave-pool>
        <lifo>true</lifo>
        <maxActive>24</maxActive>
        <maxIdle>10</maxIdle>
        <maxWait>150000</maxWait>
        <minEvictableIdleTimeMillis>100000</minEvictableIdleTimeMillis>
        <minIdle>4</minIdle>
        <testOnBorrow>true</testOnBorrow>
        <testOnReturn>false</testOnReturn>
    </slave-pool>
    <master-pool>
        <lifo>true</lifo>
        <maxActive>20</maxActive>
        <maxIdle>6</maxIdle>
        <maxWait>150000</maxWait>
        <minEvictableIdleTimeMillis>100000</minEvictableIdleTimeMillis>
        <minIdle>0</minIdle>
        <testOnBorrow>false</testOnBorrow>
        <testOnReturn>false</testOnReturn>
    </master-pool>
</nedis>
```

1. masterName

   指定 sentinel 监控的 redis-server name。redis sentinel 集群实际情况下可以监控不止一组 redis-server。为了区分不同组 server，sentinel 在启动时可以指定 server name。在 sentinel 启动时需要一个 sentinel.conf 文件，其中需要指定 sentinel monitor。如：sentinel monitor master 10.16.40.61 8800 2。其中第三个参数即为 masterName。如果不确定 masterName 是什么，可以联系 redis 集群的管理人员，查询 redis server 对应 sentinel 所配置的 masterName。

2. slave BackUp

   nedis-sentinel 一个人性化的地方是当我们进行增删改查操作时，不需要关心操作的目标 server 是 master 还是 slave。这些操作对用户是透明的。之所以这样设计是因为 redis 是 master-slave 架构，如果在 slave 上进行 write 操作，它的数据无法分发到其他 slave 或者 master，当下次 slave 进行 replication 时，这部分数据就会丢失，所以 redis 规定，write 操作必须在 master 进行，read 操作可以在 master 也可以在 slave 上。

   一般 nedis-sentinel 默认的 read 操作在 slave 上进行，但是在一些极端情况下，需要使用 master 进行 read 操作，这个时候，当 slaveBackUp 为 true 后，所用的操作都会落到 master 上，slave 只起到了备份的作用。

3. reflashWaitTimeMillis

   设置监控 slave health check 的心跳时间，单位 ms。数值越小，检测越频繁，代价越大。

4. sentinel

   指定 sentinel 的地址，至少需要一台 sentinel 的地址，为了防止单点失败的情况，sentinel 的数量最好大于 1。

   1) host

      指定 sentinel 的 ip 地址。

   2) post

指定 sentinel 的监听端口。

5. slave-pool

nedis-sentinel 具有池化特性，所以需要对每个 pool 进行一些必要的参数配置。nedis-sentinel 内置有两个 pool，slave-pool 中存放了 redis 集群中所有的 slave 连接对象。nedis-sentinel 的 pool 通过 apache commons-pool2.2.0 进行管理，参数的含义可以参考 commons-pool2.2.0 的官方文档。

6. master-pool。

Master-pool 存放了 redis 集群中 master 的连接对象。参数含义与 slave-pool 一致。

补：对于 slave-pool 和 master-pool 的参数设置需要根据具体的需求进行设置。当 nedis-sentinel 用于高并发的(近)实时服务系统时，需要将部分参数适当的调整，以免 pool 耗尽，影响(近)实时服务。

Coding

当前两步都准备就绪后，就可以进行 coding。以下是一次简单的 query 查询：

```
import com.newegg.ec.nedis.biz.Nedis;

public class MainTest {

    public static void main(String[] args) {
        Nedis nedis = null;
        try {
            nedis = new Nedis();
//          nedis = new Nedis(Mode.ReadOnly);
//          nedis = new Nedis(Mode.WriteOnly);
//          nedis = new Nedis(Mode.ReadWrite);
            System.out.println(nedis.getString(40, "1003 MONITOR USA"));
        } finally {
            nedis.returnResource();
        }
        Nedis.destroyPool();
    }
}
```

1. Nedis 构造

Nedis 是 nedis-sentinel 主要 api。Nedis 中封装了对 redis 的主要操作，有两个构造函数。默认的构造函数和带 Mode 参数的构造函数。

Mode 对象用于指定对 redis 操作的类型，是一个枚举类型，有 Mode.ReadOny，Mode.WriteOnly,Mode.ReadWrite 三种值。默认的构造函数与 Mode.ReadWrite 具有相同的作用。当用户操作中只包含一种操作，请尽量使用带参数的构造函数来减少从 pool 中获取对象所消耗的开销，对于访问量不大，性能要求不高的情况，从 pool 中获取对象的开销很小，可以直接使用默认的构造函数。

当获取对象后，就可以进行相应的操作，在使用完成后，进行 Nedis 资源的回收，请保证 nedis.returnResouce() 总是在 finally 语句块中执行，保证资源的释放，以免资源被无限的占用，导致资源的耗尽。

当程序退出时，调用 Nedis.destroyPool()方法，将 pool 资源进行回收。对于 web 服务，如 tomcat，则可以不需要调用此方法。

Nedis 对象不是一个线程安全的对象，原因是每个 nedis 对象中，至少包含了一个 socket 的连接对象，在 java api 中 socket 对象并非线程安全的，所以对于多线程的操作需要额外小心。以下代码是 nedis-sentinel 推荐的编写方式，即在每个线程中构造 nedis 对象，其实 new Nedis()的操作只是从 pool 中获取 socket 连接对象，并不真正进行 socket 的创建，所以几乎不存在什么构造上的代价。

```
public void threadTest() throws InterruptedException {
    ExecutorService exe = Executors.newFixedThreadPool(20);
    for(int j=0;j<40000;j++){
        final int k = j;
            exe.execute(new Runnable() {
                @Override
                public void run() {
                    Nedis nedis = null;
                    try {
                        nedis = new Nedis();
                        String keyValue = String.valueOf(k);
                        System.out.println(nedis.addString(9, keyValue, keyValue));
                    } catch (Exception e) {
                        throw new NedisException(e);
                    } finally {
                        if(nedis != null) {
                            nedis.returnResource();
                        }
                    }

                }
            });
    }
    Nedis.destroyPool();
}
```

2. Nedis Api

当获取到 Nedis 对象后，便可以对 redis 进行增删改查操作，nedis 屏蔽了 redis master slave 的差异，并对批量操作进行的很好的封装，几乎每个 api 都是调用 redis 的 pipeline 实现的，使 api 的效率得到的很大的提升。

下图罗列了所有 nedis 对 redis 的操作，由于时间的关系，只是对最常用的 api 进行的封装，几乎所有的 api 都能从名字中能识别出其作用。

如果有需要的 redis 操作没有在 nedis 中得到封装，欢迎大家提出。本人也非常希望大家能参与到 nedis-sentinel 的项目中，对 nedis-sentinel 进行不断的完善。

nedis-sentinel 项目是为 java search service 开发的 redis client，现在也陆续在 realtime service 和其他项目中进行使用。本人希望大家都能通过使用 nedis-sentinel 后，能从它诸多特性中受益，简化对 redis 客户端的操作复杂性。nedis-sentinel 的 svn 地址：

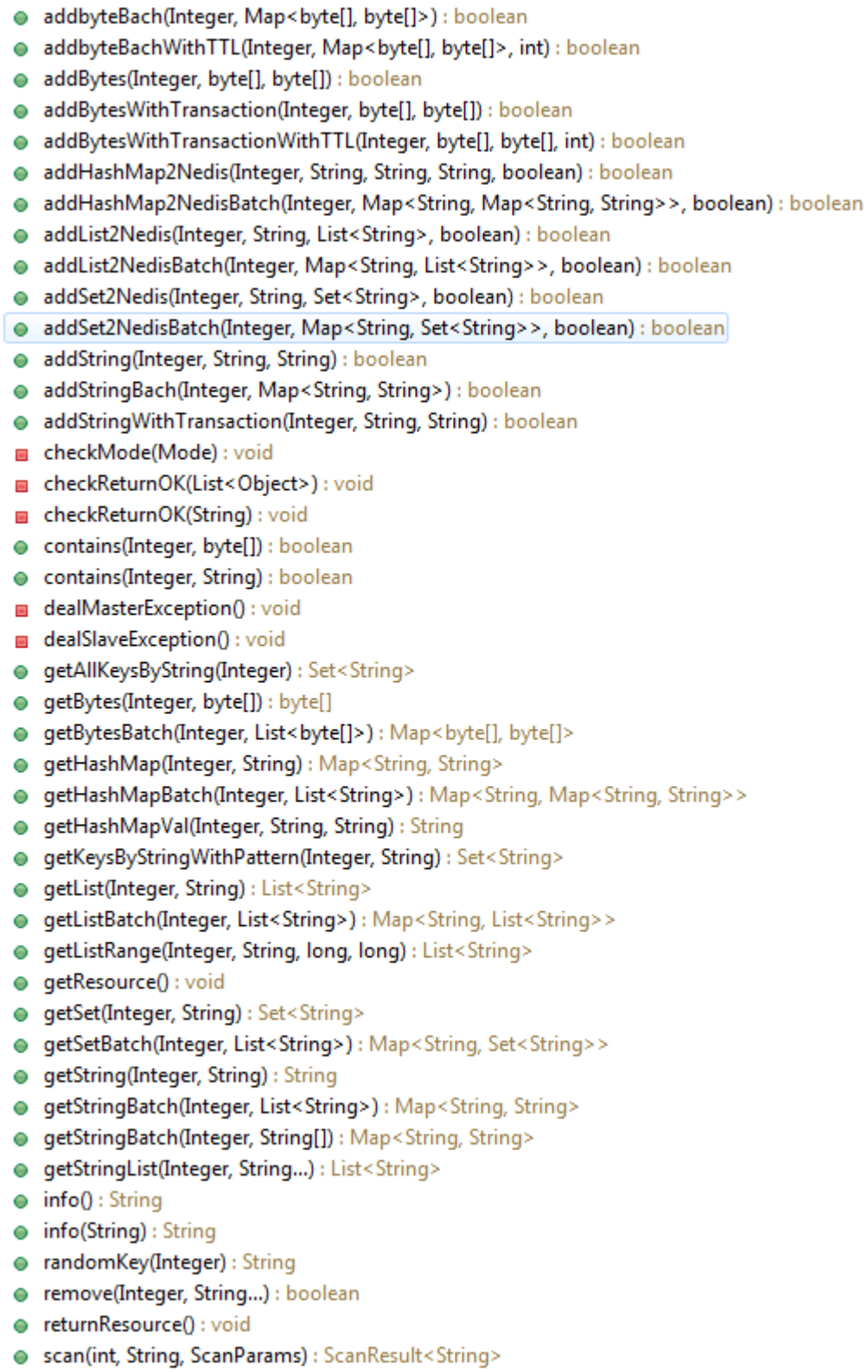
http://ssozzogqc03.buyabs.corp:9090/svn/solr-app/other-apps/SearchService/03_Code/trunk/nedis-sentinel

欢迎大家拉分支进行补充。

文档维护人员：Lyn.J.Zhang

开发人员:Lyn.J.Zhang

Support:Lyn.J.Zhang


addbyteBach(Integer, Map<byte[], byte[]>) : boolean
addbyteBachWithTTL(Integer, Map<byte[], byte[]>, int) : boolean
addBytes(Integer, byte[], byte[]) : boolean
addBytesWithTransaction(Integer, byte[], byte[]) : boolean
addBytesWithTransactionWithTTL(Integer, byte[], byte[], int) : boolean
addHashMap2Nedis(Integer, String, String, String, boolean) : boolean
addHashMap2NedisBatch(Integer, Map<String, Map<String, String>>, boolean) : boolean
addList2Nedis(Integer, String, List<String>, boolean) : boolean
addList2NedisBatch(Integer, Map<String, List<String>>, boolean) : boolean
addSet2Nedis(Integer, String, Set<String>, boolean) : boolean
addSet2NedisBatch(Integer, Map<String, Set<String>>, boolean) : boolean
addString(Integer, String, String) : boolean
addStringBach(Integer, Map<String, String>) : boolean
addStringWithTransaction(Integer, String, String) : boolean
checkMode(Mode) : void
checkReturnOK(List<Object>) : void
checkReturnOK(String) : void
contains(Integer, byte[]) : boolean
contains(Integer, String) : boolean
dealMasterException() : void
dealSlaveException() : void
getAllKeysByString(Integer) : Set<String>
getBytes(Integer, byte[]) : byte[]
getBytesBatch(Integer, List<byte[]>) : Map<byte[], byte[]>
getHashMap(Integer, String) : Map<String, String>
getHashMapBatch(Integer, List<String>) : Map<String, Map<String, String>>
getHashMapVal(Integer, String, String) : String
getKeysByStringWithPattern(Integer, String) : Set<String>
getList(Integer, String) : List<String>
getListBatch(Integer, List<String>) : Map<String, List<String>>
getListRange(Integer, String, long, long) : List<String>
getResource() : void
getSet(Integer, String) : Set<String>
getSetBatch(Integer, List<String>) : Map<String, Set<String>>
getString(Integer, String) : String
getStringBatch(Integer, List<String>) : Map<String, String>
getStringBatch(Integer, String[]) : Map<String, String>
getStringList(Integer, String...) : List<String>
info() : String
info(String) : String
randomKey(Integer) : String
remove(Integer, String...) : boolean
returnResource() : void
scan(int, String, ScanParams) : ScanResult<String>