001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017  
018package org.apache.commons.beanutils;  
019  
020  
021import java.beans.PropertyDescriptor;  
022import java.lang.ref.Reference;  
023import java.lang.ref.SoftReference;  
024import java.util.Collection;  
025import java.util.HashMap;  
026import java.util.HashSet;  
027import java.util.Iterator;  
028import java.util.Map;  
029import java.util.Set;  
030import java.util.WeakHashMap;  
031  
032  
033/\*\*  
034 \* <p>Implementation of <code>DynaClass</code> for DynaBeans that wrap  
035 \* standard JavaBean instances.</p>  
036 \*  
037 \* <p>  
038 \* It is suggested that this class should not usually need to be used directly  
039 \* to create new <code>WrapDynaBean</code> instances.  
040 \* It's usually better to call the <code>WrapDynaBean</code> constructor directly.  
041 \* For example:</p>  
042 \* <code><pre>  
043 \* Object javaBean = ...;  
044 \* DynaBean wrapper = new WrapDynaBean(javaBean);  
045 \* </pre></code>  
046 \* <p>  
047 \*  
048 \* @version $Id$  
049 \*/  
050  
051public class WrapDynaClass implements DynaClass {  
052  
053  
054 // ----------------------------------------------------------- Constructors  
055  
056  
057 /\*\*  
058 \* Construct a new WrapDynaClass for the specified JavaBean class. This  
059 \* constructor is private; WrapDynaClass instances will be created as  
060 \* needed via calls to the <code>createDynaClass(Class)</code> method.  
061 \*  
062 \* @param beanClass JavaBean class to be introspected around  
063 \* @param propUtils the {@code PropertyUtilsBean} associated with this class  
064 \*/  
065 private WrapDynaClass(final Class<?> beanClass, final PropertyUtilsBean propUtils) {  
066  
067 this.beanClassRef = new SoftReference<Class<?>>(beanClass);  
068 this.beanClassName = beanClass.getName();  
069 propertyUtilsBean = propUtils;  
070 introspect();  
071  
072 }  
073  
074  
075 // ----------------------------------------------------- Instance Variables  
076  
077 /\*\*  
078 \* Name of the JavaBean class represented by this WrapDynaClass.  
079 \*/  
080 private String beanClassName = null;  
081  
082 /\*\*  
083 \* Reference to the JavaBean class represented by this WrapDynaClass.  
084 \*/  
085 private Reference<Class<?>> beanClassRef = null;  
086  
087 /\*\* Stores the associated {@code PropertyUtilsBean} instance. \*/  
088 private final PropertyUtilsBean propertyUtilsBean;  
089  
090 /\*\*  
091 \* The JavaBean <code>Class</code> which is represented by this  
092 \* <code>WrapDynaClass</code>.  
093 \*  
094 \* @deprecated No longer initialized, use getBeanClass() method instead  
095 \*/  
096 @Deprecated  
097 protected Class<?> beanClass = null;  
098  
099  
100 /\*\*  
101 \* The set of PropertyDescriptors for this bean class.  
102 \*/  
103 protected PropertyDescriptor[] descriptors = null;  
104  
105  
106 /\*\*  
107 \* The set of PropertyDescriptors for this bean class, keyed by the  
108 \* property name. Individual descriptor instances will be the same  
109 \* instances as those in the <code>descriptors</code> list.  
110 \*/  
111 protected HashMap<String, PropertyDescriptor> descriptorsMap = new HashMap<String, PropertyDescriptor>();  
112  
113  
114 /\*\*  
115 \* The set of dynamic properties that are part of this DynaClass.  
116 \*/  
117 protected DynaProperty[] properties = null;  
118  
119  
120 /\*\*  
121 \* The set of dynamic properties that are part of this DynaClass,  
122 \* keyed by the property name. Individual descriptor instances will  
123 \* be the same instances as those in the <code>properties</code> list.  
124 \*/  
125 protected HashMap<String, DynaProperty> propertiesMap = new HashMap<String, DynaProperty>();  
126  
127  
128 // ------------------------------------------------------- Static Variables  
129  
130  
131 private static final ContextClassLoaderLocal<Map<CacheKey, WrapDynaClass>> CLASSLOADER\_CACHE =  
132 new ContextClassLoaderLocal<Map<CacheKey, WrapDynaClass>>() {  
133 @Override  
134 protected Map<CacheKey, WrapDynaClass> initialValue() {  
135 return new WeakHashMap<CacheKey, WrapDynaClass>();  
136 }  
137 };  
138  
139 /\*\*  
140 \* Get the wrap dyna classes cache. Note: This method only exists to  
141 \* satisfy the deprecated {@code dynaClasses} hash map.  
142 \*/  
143 @SuppressWarnings("unchecked")  
144 private static Map<Object, Object> getDynaClassesMap() {  
145 @SuppressWarnings("rawtypes")  
146 final  
147 Map cache = CLASSLOADER\_CACHE.get();  
148 return cache;  
149 }  
150  
151 /\*\*  
152 \* Returns the cache for the already created class instances. For each  
153 \* combination of bean class and {@code PropertyUtilsBean} instance an  
154 \* entry is created in the cache.  
155 \* @return the cache for already created {@code WrapDynaClass} instances  
156 \*/  
157 private static Map<CacheKey, WrapDynaClass> getClassesCache() {  
158 return CLASSLOADER\_CACHE.get();  
159 }  
160  
161 /\*\*  
162 \* The set of <code>WrapDynaClass</code> instances that have ever been  
163 \* created, keyed by the underlying bean Class. The keys to this map  
164 \* are Class objects, and the values are corresponding WrapDynaClass  
165 \* objects.  
166 \* <p>  
167 \* This static variable is safe even when this code is deployed via a  
168 \* shared classloader because it is keyed via a Class object. The same  
169 \* class loaded via two different classloaders will result in different  
170 \* entries in this map.  
171 \* <p>  
172 \* Note, however, that this HashMap can result in a memory leak. When  
173 \* this class is in a shared classloader it will retain references to  
174 \* classes loaded via a webapp classloader even after the webapp has been  
175 \* undeployed. That will prevent the entire classloader and all the classes  
176 \* it refers to and all their static members from being freed.  
177 \*  
178 \*\*\*\*\*\*\*\*\*\*\*\*\* !!!!!!!!!!!! PLEASE NOTE !!!!!!!!!!!! \*\*\*\*\*\*\*\*\*\*\*\*\*  
179 \*  
180 \* THE FOLLOWING IS A NASTY HACK TO SO THAT BEANUTILS REMAINS BINARY  
181 \* COMPATIBLE WITH PREVIOUS RELEASES.  
182 \*  
183 \* There are two issues here:  
184 \*  
185 \* 1) Memory Issues: The static HashMap caused memory problems (See BEANUTILS-59)  
186 \* to resolve this it has been moved into a ContextClassLoaderLocal instance  
187 \* (named CLASSLOADER\_CACHE above) which holds one copy per  
188 \* ClassLoader in a WeakHashMap.  
189 \*  
190 \* 2) Binary Compatibility: As the "dynaClasses" static HashMap is "protected"  
191 \* removing it breaks BeanUtils binary compatibility with previous versions.  
192 \* To resolve this all the methods have been overriden to delegate to the  
193 \* Map for the ClassLoader in the ContextClassLoaderLocal.  
194 \*  
195 \* @deprecated The dynaClasses Map will be removed in a subsequent release  
196 \*/  
197 @Deprecated  
198 protected static HashMap<Object, Object> dynaClasses = new HashMap<Object, Object>() {  
199 @Override  
200 public void clear() {  
201 getDynaClassesMap().clear();  
202 }  
203 @Override  
204 public boolean containsKey(final Object key) {  
205 return getDynaClassesMap().containsKey(key);  
206 }  
207 @Override  
208 public boolean containsValue(final Object value) {  
209 return getDynaClassesMap().containsValue(value);  
210 }  
211 @Override  
212 public Set<Map.Entry<Object, Object>> entrySet() {  
213 return getDynaClassesMap().entrySet();  
214 }  
215 @Override  
216 public boolean equals(final Object o) {  
217 return getDynaClassesMap().equals(o);  
218 }  
219 @Override  
220 public Object get(final Object key) {  
221 return getDynaClassesMap().get(key);  
222 }  
223 @Override  
224 public int hashCode() {  
225 return getDynaClassesMap().hashCode();  
226 }  
227 @Override  
228 public boolean isEmpty() {  
229 return getDynaClassesMap().isEmpty();  
230 }  
231 @Override  
232 public Set<Object> keySet() {  
233 // extract the classes from the key to stay backwards compatible  
234 final Set<Object> result = new HashSet<Object>();  
235 for (final CacheKey k : getClassesCache().keySet()) {  
236 result.add(k.beanClass);  
237 }  
238 return result;  
239 }  
240 @Override  
241 public Object put(final Object key, final Object value) {  
242 return getClassesCache().put(  
243 new CacheKey((Class<?>) key, PropertyUtilsBean.getInstance()),  
244 (WrapDynaClass) value);  
245 }  
246 @Override  
247 public void putAll(final Map<? extends Object, ? extends Object> m) {  
248 for (final Map.Entry<? extends Object, ? extends Object> e : m.entrySet()) {  
249 put(e.getKey(), e.getValue());  
250 }  
251 }  
252 @Override  
253 public Object remove(final Object key) {  
254 return getDynaClassesMap().remove(key);  
255 }  
256 @Override  
257 public int size() {  
258 return getDynaClassesMap().size();  
259 }  
260 @Override  
261 public Collection<Object> values() {  
262 return getDynaClassesMap().values();  
263 }  
264 };  
265  
266  
267 // ------------------------------------------------------ DynaClass Methods  
268  
269 /\*\*  
270 \* Return the class of the underlying wrapped bean.  
271 \*  
272 \* @return the class of the underlying wrapped bean  
273 \* @since 1.8.0  
274 \*/  
275 protected Class<?> getBeanClass() {  
276 return beanClassRef.get();  
277 }  
278  
279 /\*\*  
280 \* Return the name of this DynaClass (analogous to the  
281 \* <code>getName()</code> method of <code>java.lang.Class</code), which  
282 \* allows the same <code>DynaClass</code> implementation class to support  
283 \* different dynamic classes, with different sets of properties.  
284 \*  
285 \* @return the name of the DynaClass  
286 \*/  
287 public String getName() {  
288  
289 return beanClassName;  
290  
291 }  
292  
293  
294 /\*\*  
295 \* Return a property descriptor for the specified property, if it exists;  
296 \* otherwise, return <code>null</code>.  
297 \*  
298 \* @param name Name of the dynamic property for which a descriptor  
299 \* is requested  
300 \* @return The descriptor for the specified property  
301 \*  
302 \* @throws IllegalArgumentException if no property name is specified  
303 \*/  
304 public DynaProperty getDynaProperty(final String name) {  
305  
306 if (name == null) {  
307 throw new IllegalArgumentException  
308 ("No property name specified");  
309 }  
310 return (propertiesMap.get(name));  
311  
312 }  
313  
314  
315 /\*\*  
316 \* <p>Return an array of <code>ProperyDescriptors</code> for the properties  
317 \* currently defined in this DynaClass. If no properties are defined, a  
318 \* zero-length array will be returned.</p>  
319 \*  
320 \* <p><strong>FIXME</strong> - Should we really be implementing  
321 \* <code>getBeanInfo()</code> instead, which returns property descriptors  
322 \* and a bunch of other stuff?</p>  
323 \*  
324 \* @return the set of properties for this DynaClass  
325 \*/  
326 public DynaProperty[] getDynaProperties() {  
327  
328 return (properties);  
329  
330 }  
331  
332  
333 /\*\*  
334 \* <p>Instantiates a new standard JavaBean instance associated with  
335 \* this DynaClass and return it wrapped in a new WrapDynaBean  
336 \* instance. <strong>NOTE</strong> the JavaBean should have a  
337 \* no argument constructor.</p>  
338 \*  
339 \* <strong>NOTE</strong> - Most common use cases should not need to use  
340 \* this method. It is usually better to create new  
341 \* <code>WrapDynaBean</code> instances by calling its constructor.  
342 \* For example:</p>  
343 \* <code><pre>  
344 \* Object javaBean = ...;  
345 \* DynaBean wrapper = new WrapDynaBean(javaBean);  
346 \* </pre></code>  
347 \* <p>  
348 \* (This method is needed for some kinds of <code>DynaBean</code> framework.)  
349 \* </p>  
350 \*  
351 \* @return A new <code>DynaBean</code> instance  
352 \* @throws IllegalAccessException if the Class or the appropriate  
353 \* constructor is not accessible  
354 \* @throws InstantiationException if this Class represents an abstract  
355 \* class, an array class, a primitive type, or void; or if instantiation  
356 \* fails for some other reason  
357 \*/  
358 public DynaBean newInstance()  
359 throws IllegalAccessException, InstantiationException {  
360  
361 return new WrapDynaBean(getBeanClass().newInstance());  
362  
363 }  
364  
365  
366 // --------------------------------------------------------- Public Methods  
367  
368  
369 /\*\*  
370 \* Return the PropertyDescriptor for the specified property name, if any;  
371 \* otherwise return <code>null</code>.  
372 \*  
373 \* @param name Name of the property to be retrieved  
374 \* @return The descriptor for the specified property  
375 \*/  
376 public PropertyDescriptor getPropertyDescriptor(final String name) {  
377  
378 return (descriptorsMap.get(name));  
379  
380 }  
381  
382  
383 // --------------------------------------------------------- Static Methods  
384  
385  
386 /\*\*  
387 \* Clear our cache of WrapDynaClass instances.  
388 \*/  
389 public static void clear() {  
390  
391 getClassesCache().clear();  
392  
393 }  
394  
395  
396 /\*\*  
397 \* Create (if necessary) and return a new <code>WrapDynaClass</code>  
398 \* instance for the specified bean class.  
399 \*  
400 \* @param beanClass Bean class for which a WrapDynaClass is requested  
401 \* @return A new <i>Wrap</i> {@link DynaClass}  
402 \*/  
403 public static WrapDynaClass createDynaClass(final Class<?> beanClass) {  
404  
405 return createDynaClass(beanClass, null);  
406  
407 }  
408  
409  
410 /\*\*  
411 \* Create (if necessary) and return a new {@code WrapDynaClass} instance  
412 \* for the specified bean class using the given {@code PropertyUtilsBean}  
413 \* instance for introspection. Using this method a specially configured  
414 \* {@code PropertyUtilsBean} instance can be hooked into the introspection  
415 \* mechanism of the managed bean. The argument is optional; if no  
416 \* {@code PropertyUtilsBean} object is provided, the default instance is used.  
417 \* @param beanClass Bean class for which a WrapDynaClass is requested  
418 \* @param pu the optional {@code PropertyUtilsBean} to be used for introspection  
419 \* @return A new <i>Wrap</i> {@link DynaClass}  
420 \* @since 1.9  
421 \*/  
422 public static WrapDynaClass createDynaClass(final Class<?> beanClass, final PropertyUtilsBean pu) {  
423  
424 final PropertyUtilsBean propUtils = (pu != null) ? pu : PropertyUtilsBean.getInstance();  
425 final CacheKey key = new CacheKey(beanClass, propUtils);  
426 WrapDynaClass dynaClass = getClassesCache().get(key);  
427 if (dynaClass == null) {  
428 dynaClass = new WrapDynaClass(beanClass, propUtils);  
429 getClassesCache().put(key, dynaClass);  
430 }  
431 return (dynaClass);  
432  
433 }  
434  
435  
436 // ------------------------------------------------------ Protected Methods  
437  
438 /\*\*  
439 \* Returns the {@code PropertyUtilsBean} instance associated with this class. This  
440 \* bean is used for introspection.  
441 \*  
442 \* @return the associated {@code PropertyUtilsBean} instance  
443 \* @since 1.9  
444 \*/  
445 protected PropertyUtilsBean getPropertyUtilsBean() {  
446 return propertyUtilsBean;  
447 }  
448  
449 /\*\*  
450 \* Introspect our bean class to identify the supported properties.  
451 \*/  
452 protected void introspect() {  
453  
454 // Look up the property descriptors for this bean class  
455 final Class<?> beanClass = getBeanClass();  
456 PropertyDescriptor[] regulars =  
457 getPropertyUtilsBean().getPropertyDescriptors(beanClass);  
458 if (regulars == null) {  
459 regulars = new PropertyDescriptor[0];  
460 }  
461 @SuppressWarnings("deprecation")  
462 Map<?, ?> mappeds =  
463 PropertyUtils.getMappedPropertyDescriptors(beanClass);  
464 if (mappeds == null) {  
465 mappeds = new HashMap<Object, Object>();  
466 }  
467  
468 // Construct corresponding DynaProperty information  
469 properties = new DynaProperty[regulars.length + mappeds.size()];  
470 for (int i = 0; i < regulars.length; i++) {  
471 descriptorsMap.put(regulars[i].getName(),  
472 regulars[i]);  
473 properties[i] =  
474 new DynaProperty(regulars[i].getName(),  
475 regulars[i].getPropertyType());  
476 propertiesMap.put(properties[i].getName(),  
477 properties[i]);  
478 }  
479 int j = regulars.length;  
480 final Iterator<?> names = mappeds.keySet().iterator();  
481 while (names.hasNext()) {  
482 final String name = (String) names.next();  
483 final PropertyDescriptor descriptor =  
484 (PropertyDescriptor) mappeds.get(name);  
485 properties[j] =  
486 new DynaProperty(descriptor.getName(),  
487 Map.class);  
488 propertiesMap.put(properties[j].getName(),  
489 properties[j]);  
490 j++;  
491 }  
492  
493 }  
494  
495 /\*\*  
496 \* A class representing the combined key for the cache of {@code WrapDynaClass}  
497 \* instances. A single key consists of a bean class and an instance of  
498 \* {@code PropertyUtilsBean}. Instances are immutable.  
499 \*/  
500 private static class CacheKey {  
501 /\*\* The bean class. \*/  
502 private final Class<?> beanClass;  
503  
504 /\*\* The instance of PropertyUtilsBean. \*/  
505 private final PropertyUtilsBean propUtils;  
506  
507 /\*\*  
508 \* Creates a new instance of {@code CacheKey}.  
509 \*  
510 \* @param beanCls the bean class  
511 \* @param pu the instance of {@code PropertyUtilsBean}  
512 \*/  
513 public CacheKey(final Class<?> beanCls, final PropertyUtilsBean pu) {  
514 beanClass = beanCls;  
515 propUtils = pu;  
516 }  
517  
518 @Override  
519 public int hashCode() {  
520 final int factor = 31;  
521 int result = 17;  
522 result = factor \* beanClass.hashCode() + result;  
523 result = factor \* propUtils.hashCode() + result;  
524 return result;  
525 }  
526  
527 @Override  
528 public boolean equals(final Object obj) {  
529 if (this == obj) {  
530 return true;  
531 }  
532 if (!(obj instanceof CacheKey)) {  
533 return false;  
534 }  
535  
536 final CacheKey c = (CacheKey) obj;  
537 return beanClass.equals(c.beanClass) && propUtils.equals(c.propUtils);  
538 }  
539 }  
540}