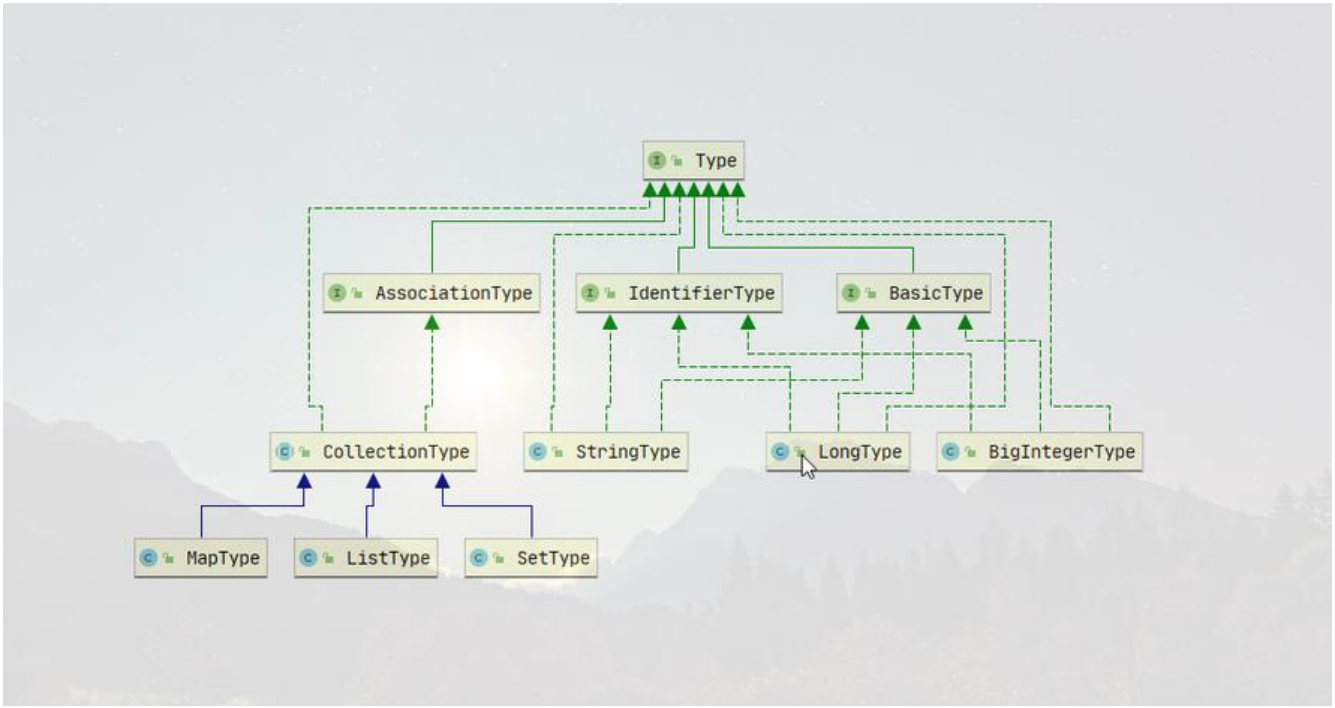
Hibernate Type 源码解析

作者：boolean-dev
原文链接：https://ld246.com/article/1600933162862
来源网站：链滴

type 的类图



AssociationType: 关联类型的 type，主要用于外键等相关

IdentifierType: 主键相关的类型 type

BasicType: 基础类型，例如 long,int,string 等基础类型

type 注册

基础类型的注册器

```
/*
 * Hibernate, Relational Persistence for Idiomatic Java
 *
 * License: GNU Lesser General Public License (LGPL), version 2.1 or later.
 * See the lgpl.txt file in the root directory or <http://www.gnu.org/licenses/lgpl-2.1.html>.
 */
package org.hibernate.type;

import java.io.Serializable;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import org.hibernate.HibernateException;
import org.hibernate.internal.CoreLogging;
import org.hibernate.internal.CoreMessageLogger;
import org.hibernate.type.spi.TypeConfiguration;
import org.hibernate.usertype.CompositeUserType;
import org.hibernate.usertype.UserType;

/**
 * A registry of {@link BasicType} instances
 *
 * @author Steve Ebersole
 */
public class BasicTypeRegistry implements Serializable {
    private static final CoreMessageLogger LOG = CoreLogging.messageLogger( BasicTypeReg
istry.class );

    // TODO : analyze these sizing params; unfortunately this seems to be the only way to give
 "concurrencyLevel"
    // 使用 ConcurrentHashMap 保证线程安全
    private Map<String, BasicType> registry = new ConcurrentHashMap<>( 100, .75f, 1 );
    private boolean locked;
    private TypeConfiguration typeConfiguration;

    public BasicTypeRegistry(TypeConfiguration typeConfiguration){
        this();
        this.typeConfiguration = typeConfiguration;
    }

    // 构造方法，用于注册各种对应的基础类型
    public BasicTypeRegistry() {
        register( BooleanType.INSTANCE );
        register( NumericBooleanType.INSTANCE );
        register( TrueFalseType.INSTANCE );
        register( YesNoType.INSTANCE );

        register( ByteType.INSTANCE );
        register( CharacterType.INSTANCE );
        register( ShortType.INSTANCE );
        register( IntegerType.INSTANCE );
        register( LongType.INSTANCE );
```

```
register( FloatType.INSTANCE );
register( DoubleType.INSTANCE );
register( BigDecimalType.INSTANCE );
register( BigIntegerType.INSTANCE );

register( StringType.INSTANCE );
register( StringNVarcharType.INSTANCE );
register( CharacterNCharType.INSTANCE );
register( UrlType.INSTANCE );

register( DurationType.INSTANCE );
register( InstantType.INSTANCE );
register( LocalDateTimeType.INSTANCE );
register( LocalDateType.INSTANCE );
register( LocalTimeType.INSTANCE );
register( OffsetDateTimeType.INSTANCE );
register( OffsetTimeType.INSTANCE );
register( ZonedDateTimeType.INSTANCE );

register( DateType.INSTANCE );
register( TimeType.INSTANCE );
register( TimestampType.INSTANCE );
register( DbTimestampType.INSTANCE );
register( CalendarType.INSTANCE );
register( CalendarDateType.INSTANCE );
register( CalendarTimeType.INSTANCE );

register( LocaleType.INSTANCE );
register( CurrencyType.INSTANCE );
register( TimeZoneType.INSTANCE );
register( ClassType.INSTANCE );
register( UUIDBinaryType.INSTANCE );
register( UUIDCharType.INSTANCE );

register( BinaryType.INSTANCE );
register( WrapperBinaryType.INSTANCE );
register( RowVersionType.INSTANCE );
register( ImageType.INSTANCE );
register( CharArrayType.INSTANCE );
register( CharacterArrayType.INSTANCE );
register( TextType.INSTANCE );
register( NTextType.INSTANCE );
register( BlobType.INSTANCE );
register( MaterializedBlobType.INSTANCE );
register( ClobType.INSTANCE );
register( NClobType.INSTANCE );
register( MaterializedClobType.INSTANCE );
register( MaterializedNClobType.INSTANCE );
register( SerializableType.INSTANCE );

register( ObjectType.INSTANCE );

//noinspection unchecked
register( new AdaptedImmutableType( DateType.INSTANCE ) );
```

```
    //noinspection unchecked
    register( new AdaptedImmutableType( TimeType.INSTANCE ) );
    //noinspection unchecked
    register( new AdaptedImmutableType( TimestampType.INSTANCE ) );
    //noinspection unchecked
    register( new AdaptedImmutableType( DbTimestampType.INSTANCE ) );
    //noinspection unchecked
    register( new AdaptedImmutableType( CalendarType.INSTANCE ) );
    //noinspection unchecked
    register( new AdaptedImmutableType( CalendarDateType.INSTANCE ) );
    //noinspection unchecked
    register( new AdaptedImmutableType( BinaryType.INSTANCE ) );
    //noinspection unchecked
    register( new AdaptedImmutableType( SerializableType.INSTANCE ) );
}

/**
 * Constructor version used during shallow copy
 *
 * @param registeredTypes The type map to copy over
 */
@SuppressWarnings({"UnusedDeclaration"})
private BasicTypeRegistry(Map<String, BasicType> registeredTypes) {
    registry.putAll( registeredTypes );
    locked = true;
}

// 注册对应的基础类型
public void register(BasicType type) {
    register( type, type.getRegistrationKeys() );
}

// 注册对应的基础类型最终实现方法
public void register(BasicType type, String[] keys) {

    // 判断是否加锁
    if ( locked ) {
        throw new HibernateException( "Can not alter TypeRegistry at this time" );
    }

    // 判断 type 是否为空
    if ( type == null ) {
        throw new HibernateException( "Type to register cannot be null" );
    }

    if ( keys == null || keys.length == 0 ) {
        LOG.typeDefinedNoRegistrationKeys( type );
        return;
    }

    for ( String key : keys ) {
        // be safe...
        if ( key == null ) {
            continue;
```

```
            }
            //Use String#intern here as there's high chances of duplicates combined with long te
m usage:
            //just running our testsuite would generate 210,000 instances for the String "java.lang
Class" alone.
            //Incidentally this might help with map lookup efficiency too.
            key = key.intern();
            LOG.debugf( "Adding type registration %s -> %s", key, type );
            final Type old = registry.put( key, type );
            if ( old != null && old != type ) {
                LOG.typeRegistrationOverridesPrevious( key, old );
            }
        }
    }

    public void register(UserType type, String[] keys) {
        register( new CustomType( type, keys ) );
    }

    public void register(CompositeUserType type, String[] keys) {
        register( new CompositeCustomType( type, keys ) );
    }

    public void unregister(String... keys) {
        for ( String key : keys ) {
            registry.remove( key );
        }
    }

    public BasicType getRegisteredType(String key) {
        return registry.get( key );
    }

    public BasicTypeRegistry shallowCopy() {
        return new BasicTypeRegistry( this.registry );
    }
}
```

从 TypeFactory 中获取 type

```
/*
 * Hibernate, Relational Persistence for Idiomatic Java
 *
 * License: GNU Lesser General Public License (LGPL), version 2.1 or later.
 * See the lgpl.txt file in the root directory or <http://www.gnu.org/licenses/lgpl-2.1.html>.
 */
package org.hibernate.type;

import java.io.Serializable;
import java.util.Properties;

import org.hibernate.MappingException;
import org.hibernate.boot.registry.classloading.spi.ClassLoaderService;
import org.hibernate.boot.registry.classloading.spi.ClassLoadingException;
```

```
import org.hibernate.type.spi.TypeConfiguration;
import org.hibernate.usertype.CompositeUserType;
import org.hibernate.usertype.UserType;

/**
 * Acts as the contract for getting types and as the mediator between {@link BasicTypeRegistr
} and {@link TypeFactory}.
 *
 * @author Steve Ebersole
 *
 * @deprecated (since 5.3) No replacement, access to and handling of Types will be much diff
rent in 6.0
 */
@Deprecated
public class TypeResolver implements Serializable {
    private final TypeFactory typeFactory;
    private final TypeConfiguration typeConfiguration;

    public TypeResolver(TypeConfiguration typeConfiguration, TypeFactory typeFactory){
        this.typeConfiguration = typeConfiguration;
        this.typeFactory = typeFactory;
    }

//    public TypeResolver() {
//        this( new BasicTypeRegistry(), new TypeFactory() );
//    }
//
//    /**
//     * @deprecated (since 5.3)
//     */
//    @Deprecated
//    public TypeResolver(BasicTypeRegistry basicTypeRegistry, TypeFactory typeFactory) {
//        this.basicTypeRegistry = basicTypeRegistry;
//        this.typeFactory = typeFactory;
//    }

//    public TypeResolver scope(SessionFactoryImplementor factory) {
//        typeFactory.injectSessionFactory( factory );
//        return new TypeResolver( basicTypeRegistry.shallowCopy(), typeFactory );
//    }

    public void registerTypeOverride(BasicType type) {
        typeConfiguration.getBasicTypeRegistry().register( type );
    }

    public void registerTypeOverride(UserType type, String[] keys) {
        typeConfiguration.getBasicTypeRegistry().register( type, keys );
    }

    public void registerTypeOverride(CompositeUserType type, String[] keys) {
        typeConfiguration.getBasicTypeRegistry().register( type, keys );
    }

    public TypeFactory getTypeFactory() {
```

```
        return typeFactory;
    }

    /**
     * Locate a Hibernate {@linkplain BasicType basic type} given (one of) its registration names
     *
     * @param name The registration name
     *
     * @return The registered type
     */
    public BasicType basic(String name) {
        return typeConfiguration.getBasicTypeRegistry().getRegisteredType( name );
    }

    /**
     * See {@link #heuristicType(String, Properties)}
     *
     * @param typeName The name (see heuristic algorithm discussion on {@link #heuristicTy
e(String, Properties)}).
     *
     * @return The deduced type; may be null.
     *
     * @throws MappingException Can be thrown from {@link #heuristicType(String, Properties
}
     */
    public Type heuristicType(String typeName) throws MappingException {
        return heuristicType( typeName, null );
    }

    /**
     * Uses heuristics to deduce the proper {@link Type} given a string naming the type or Java
class.
     * <p/>
     * The search goes as follows:<ol>
     *     <li>search for a basic type with 'typeName' as a registration key</li>
     *     <li>
     *         look for 'typeName' as a class name and<ol>
     *             <li>if it names a {@link Type} implementor, return an instance</li>
     *             <li>if it names a {@link CompositeUserType} or a {@link UserType}, return an inst
nce of class wrapped into the appropriate {@link Type} adapter</li>
     *             <li>if it implements {@link org.hibernate.classic.Lifecycle}, return the correspondi
g entity type</li>
     *             <li>if it implements {@link Serializable}, return the corresponding serializable typ
</li>
     *         </ol>
     *     </li>
     * </ol>
     *
     * @param typeName The name (see heuristic algorithm above).
     * @param parameters Any parameters for the type.  Only applied if built!
     *
     * @return The deduced type; may be null.
     *
```

```
     * @throws MappingException Indicates a problem attempting to resolve 'typeName' as a
@link Class}
     */

     // 核心代码
    public Type heuristicType(String typeName, Properties parameters) throws MappingExcept
on {
        // 如果为 basicType,z则直接获取
        Type type = basic( typeName );
        if ( type != null ) {
            return type;
        }

        try {

            // 如果Basic中未获取到，通过反射加载Type
            final ClassLoaderService classLoaderService = typeConfiguration.getServiceRegistry().
etService( ClassLoaderService.class );
            Class typeClass = classLoaderService.classForName( typeName );
            if ( typeClass != null ) {
                return typeFactory.byClass( typeClass, parameters );
            }
        }
        catch ( ClassLoadingException ignore ) {
        }

        return null;
    }
}
```