5 Design Patterns in Cassandra

1. Singleton Pattern

Singleton pattern is one of the simplest and most common design patterns in Java projects. It is one of the creational patterns and provides a way to create an object. It is used when we want to limit the instantiation of one class to only one object.

In Cassandra, one of the usage is in the `AuditLogManager` class. "It is the central location for managing the logging of client/user-initiated actions (like queries, log in commands, and so on)". (copied from the comment)

The only instance is created with:

```java
private static final AuditLogManager instance = new AuditLogManager();
```

And the constructor is private, so that it cannot be instantiated from outside:

```java
private AuditLogManager()
{
    fullQueryLogger = new FullQueryLogger();

    if (DatabaseDescriptor.getAuditLoggingOptions().enabled)
    {
        logger.info("Audit logging is enabled.");
        auditLogger =
getAuditLogger(DatabaseDescriptor.getAuditLoggingOptions().logger);
        isAuditLogEnabled = true;
    }
    else
    {
        logger.debug("Audit logging is disabled.");
        isAuditLogEnabled = false;
        auditLogger = new NoOpAuditLogger();
    }

    filter =
AuditLogFilter.create(DatabaseDescriptor.getAuditLoggingOptions());
}
```

The instance can be used from the `getInstance()` method:

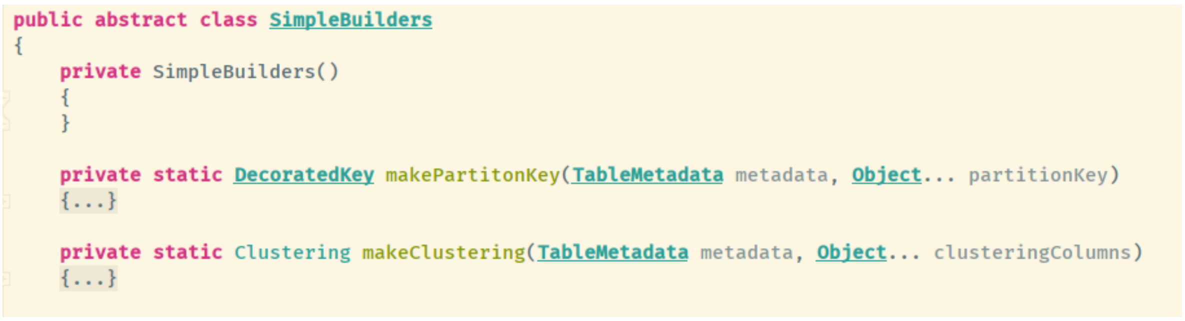
```java
public static AuditLogManager getInstance()
{
    return instance;
}
```

The singleton pattern is used here because for one client, there should be only one location to manage the query or logging actions, to avoid conflict operations and permission issues. If there could be two AuditLogManager class for one user, the user could log out at one place and make queries at another place. These actions confuse Cassandra and may provide a unsecure way to query information without logging in or having corresponding permissions, and finally leading to security issues. So, only one instance of the AuditLogManager should be created at the same time, and the singleton pattern is used properly.

2. Builder Pattern

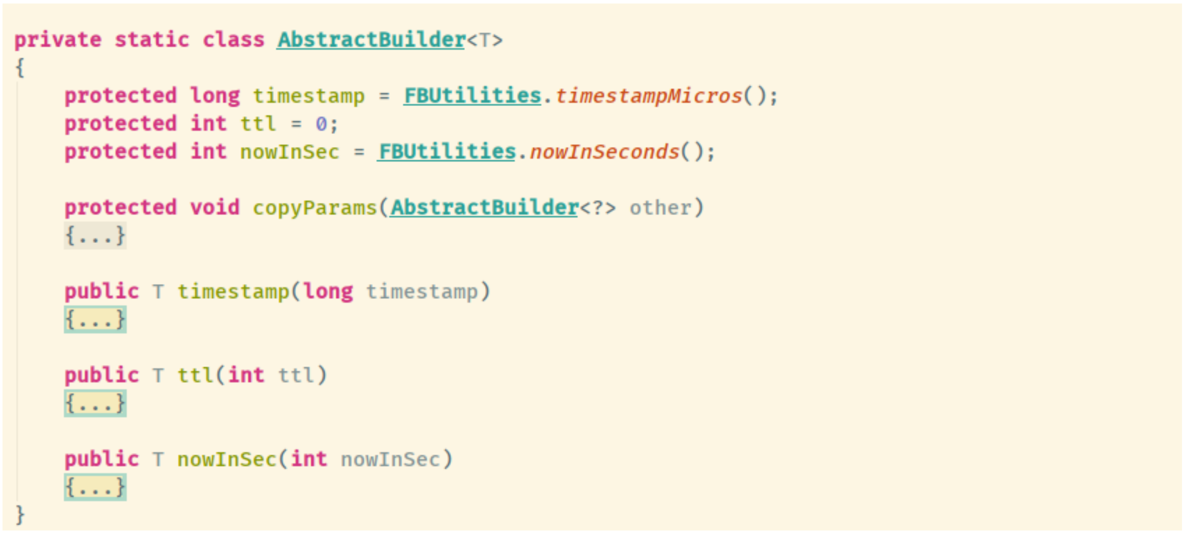
Builder pattern is another simple and common pattern in object oriented projects. It is used to separate the process of building one complex object, construct it step by step, and return the object in the last step. Different representations of one object are created using the builder pattern.

In Cassandra, the abstract class `SimpleBuilders` has only two methods: `makePartitionKey` and `makeClustering`, providing basic operations.



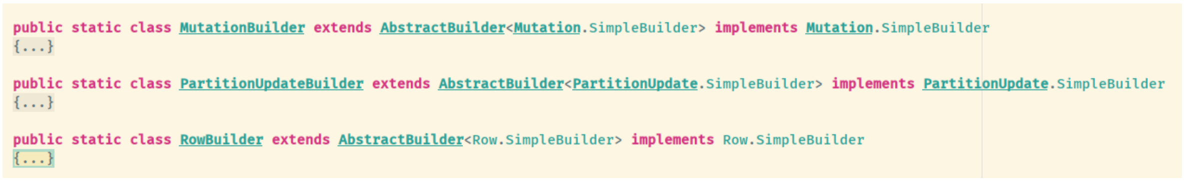
```
public abstract class SimpleBuilders
{
    private SimpleBuilders()
    {
    }

    private static DecoratedKey makePartitonKey(TableMetadata metadata, Object... partitionKey)
    {...}

    private static Clustering makeClustering(TableMetadata metadata, Object... clusteringColumns)
    {...}
```

The inner static class `AbstractBuilder` is created with four more methods and three more fields to provide more functions for the builder.



```
private static class AbstractBuilder<T>
{
    protected long timestamp = FBUtilities.timestampMicros();
    protected int ttl = 0;
    protected int nowInSec = FBUtilities.nowInSeconds();

    protected void copyParams(AbstractBuilder<?> other)
    {...}

    public T timestamp(long timestamp)
    {...}

    public T ttl(int ttl)
    {...}

    public T nowInSec(int nowInSec)
    {...}
}
```

Three builders including `MutationBuilder`, `PartitionUpdateBuilder` and `RowBuilder` are implemented extending the `AbstractBuilder`.



```
public static class MutationBuilder extends AbstractBuilder<Mutation.SimpleBuilder> implements Mutation.SimpleBuilder
{...}

public static class PartitionUpdateBuilder extends AbstractBuilder<PartitionUpdate.SimpleBuilder> implements PartitionUpdate.SimpleBuilder
{...}

public static class RowBuilder extends AbstractBuilder<Row.SimpleBuilder> implements Row.SimpleBuilder
{...}
```

In this way, the same code is reused but can construct different objects with different properties. It allows to provide the control of different steps in the constructing process and vary the internal representation. Without the builder pattern here, many construction process of the three builders will be repeated and it is hard to find the commonness and differences among them.

3. Strategy Pattern

Essentially strategy pattern encapsulates algorithms while decoupling each individual details of a behavior that implements a general interface. It is one of the most common design patterns that Cassandra uses and below is an example of it.

The `AliasedSelectable.java` class implements an interface `Selectable`, and then it creates an object Selectable as a member variable of this class, which has not been initialized at first.

```
final class AliasedSelectable implements Selectable
{
    /**
     * The selectable
     */
    private final Selectable selectable;
```

Then it creates a constructor that takes the `Selectable` as a parameter, which means whoever is creating a instance of `AliasedSelectable` class will pass a concrete implementation of the `Selectable` interface to initialize this `AliasedSelectable` class.

```
public AliasedSelectable(Selectable selectable, ColumnIdentifier alias)
    {
        this.selectable = selectable;
        this.alias = alias;
    }
```

After that, specific methods were implemented that apply to only `AliasedSelectable` class, and it is different implementation of other sub classes that realized `Selectable`

```
@Override
    public TestResult testAssignment(String keyspace, ColumnSpecification receiver)
    {
        return selectable.testAssignment(keyspace, receiver);
    }
```

```
@Override
    public AbstractType<?> getExactTypeIfKnown(String keyspace)
    {
        return selectable.getExactTypeIfKnown(keyspace);
    }

    @Override
    public boolean selectColumns(Predicate<ColumnMetadata> predicate)
    {
        return selectable.selectColumns(predicate);
    }
```

Other implementations of `Selectable` interface include inner static class `BewteenParentheseOrWithTuple` , it also takes a list of Selectables as parameter of its constructor so that different methods can be used when creating a instance through this constructor.

```java
public static class BetweenParenthesesOrWithTuple implements Selectable
    {
        /**
         * The tuple elements or the element between the parentheses
         */
        private final List<Selectable> selectables;

        public BetweenParenthesesOrWithTuple(List<Selectable> selectables)
        {
            this.selectables = selectables;
        }
```

4. Factory Method Pattern

In a factory pattern, instead of directly using a call method through a constructor, a product is created within a factory class. And the business logic will decide in what way and what concrete typle of the product will be created based on the possible concrete implementation of that factory.

Usually there will be a base factory that is either abstract class or an interface, which allow concrete subclass to implement. In Cassandra, one of the inner static class named `Factory` is playing such a role.

```java
public static abstract class Factory
```

This class provides a couple of public methods that can be called by its concrete factory class later.

```java
public ColumnSpecification getColumnSpecification(TableMetadata table)
        {
            return new ColumnSpecification(table.keyspace,
                                           table.name,
                                           new
ColumnIdentifier(getColumnName(), true), // note that the name is not
necessarily

           // a true column name so we shouldn't intern it
                                           getReturnType());
        }
```

It will return the column specification corresponding to the output value of the selector instances created by this factory. Following is multiple abstract methods that is empty on this base factory class but implemented some where else to create concrete products. One example of abstract method returns a Selector instance.

```java
public abstract Selector newInstance(QueryOptions options) throws
InvalidRequestException;
```

Concrete "selector" product is created at different subclasses and below are some examples. An `ElementSelector` is created.

```java
public Selector newInstance(QueryOptions options) throws InvalidRequestException
            {
                ByteBuffer keyValue = key.bindAndGet(options);
                if (keyValue == null)
                    throw new InvalidRequestException("Invalid null value for element selection on " + factory.getColumnName());
                if (keyValue == ByteBufferUtil.UNSET_BYTE_BUFFER)
                    throw new InvalidRequestException("Invalid unset value for element selection on " + factory.getColumnName());
                return new ElementSelector(factory.newInstance(options), keyValue);
            }
```

Another concrete instance is `SlicedSelector`:

```java
public Selector newInstance(QueryOptions options) throws InvalidRequestException
            {
                ByteBuffer fromValue = from.bindAndGet(options);
                ByteBuffer toValue = to.bindAndGet(options);
                // Note that we use UNSET values to represent no bound, so null is truly invalid
                if (fromValue == null || toValue == null)
                    throw new InvalidRequestException("Invalid null value for slice selection on " + factory.getColumnName());
                return new SliceSelector(factory.newInstance(options), from.bindAndGet(options), to.bindAndGet(options));
            }
```

A third concrete instance created is `FieldSelector`:

```java
public Selector newInstance(QueryOptions options) throws InvalidRequestException
            {
                return new FieldSelector(type, field, factory.newInstance(options));
            }
```

5. Observer Pattern

In an observer pattern, there are two main roles, subscriber and publisher. A publisher will take care of a list of all of its subscribers and will have a notify method to notify any updates of a current state. There will also be subscribe and unsubscribe functions that allow new object to add to or remove from the list of subscribers.

In Cassandra, one implementation of this pattern is subscribers will be notified by a publisher and update host scores. It is first defined as a inner interface as below:

```java
public class LatencySubscribers
{
    public interface Subscriber
    {
        void receiveTiming(InetAddressAndPort address, long latency, TimeUnit unit);
    }
```

This `LatencySubscribers` class act as a "publisher" that has public methods such as `subscribe` and `add`. It tracks latency information for dynamic snitch, who is a concrete listener/subscriber class.

```java
public class LatencySubscribers
{
    ...
    private volatile Subscriber subscribers;

    public void subscribe(Subscriber subscriber)
    {
        subscribersUpdater.accumulateAndGet(this, subscriber, LatencySubscribers::merge);
    }

    public void add(InetAddressAndPort address, long latency, TimeUnit unit)
    {
        Subscriber subscribers = this.subscribers;
        if (subscribers != null)
            subscribers.receiveTiming(address, latency, unit);
    }
}
```

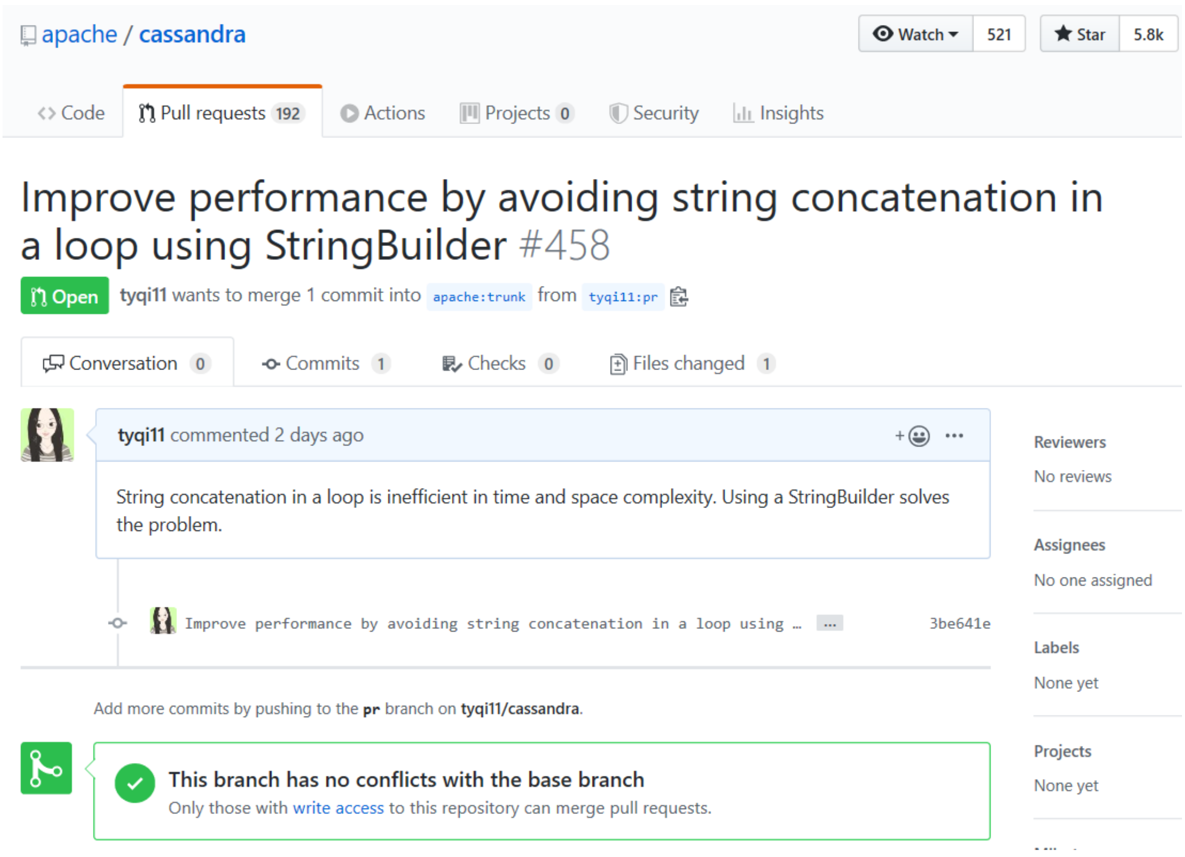
```java
/**
     * Track latency information for the dynamic snitch
     *
     * @param cb      the callback associated with this message -- this lets us know if it's a message type we're interested in
     * @param address the host that replied to the message
     */
    public void maybeAdd(RequestCallback cb, InetAddressAndPort address, long latency, TimeUnit unit)
    {
        if (cb.trackLatencyForSnitch())
            add(address, latency, unit);
    }
```

A concrete example class that implement `Subscriber` interface and realize its `receiveTiming` method is `DynamicEndpointSnitch` class. This class listens to the publisher `LatencySubscribers` and gets notified when there is a change in latency, and then update system of host scores.

```
public void receiveTiming(InetAddressAndPort host, long latency, TimeUnit
unit) // this is cheap
    {
        ExponentiallyDecayingReservoir sample = samples.get(host);
        if (sample == null)
        {
            ExponentiallyDecayingReservoir maybeNewSample = new
ExponentiallyDecayingReservoir(WINDOW_SIZE, ALPHA);
            sample = samples.putIfAbsent(host, maybeNewSample);
            if (sample == null)
                sample = maybeNewSample;
        }
        sample.update(unit.toMillis(latency));
    }
```

First Pull request

Our first pull request to Cassandra is #458.



apache / cassandra

Watch 521 Star 5.8k

Code | Pull requests 192 | Actions | Projects 0 | Security | Insights

Improve performance by avoiding string concatenation in a loop using StringBuilder #458

Open tyqi11 wants to merge 1 commit into `apache:trunk` from `tyqi11:pr`

Conversation 0 | Commits 1 | Checks 0 | Files changed 1

tyqi11 commented 2 days ago

String concatenation in a loop is inefficient in time and space complexity. Using a StringBuilder solves the problem.

`Improve performance by avoiding string concatenation in a loop using … … 3be641e`

Add more commits by pushing to the **pr** branch on **tyqi11/cassandra**.

This branch has no conflicts with the base branch
Only those with write access to this repository can merge pull requests.

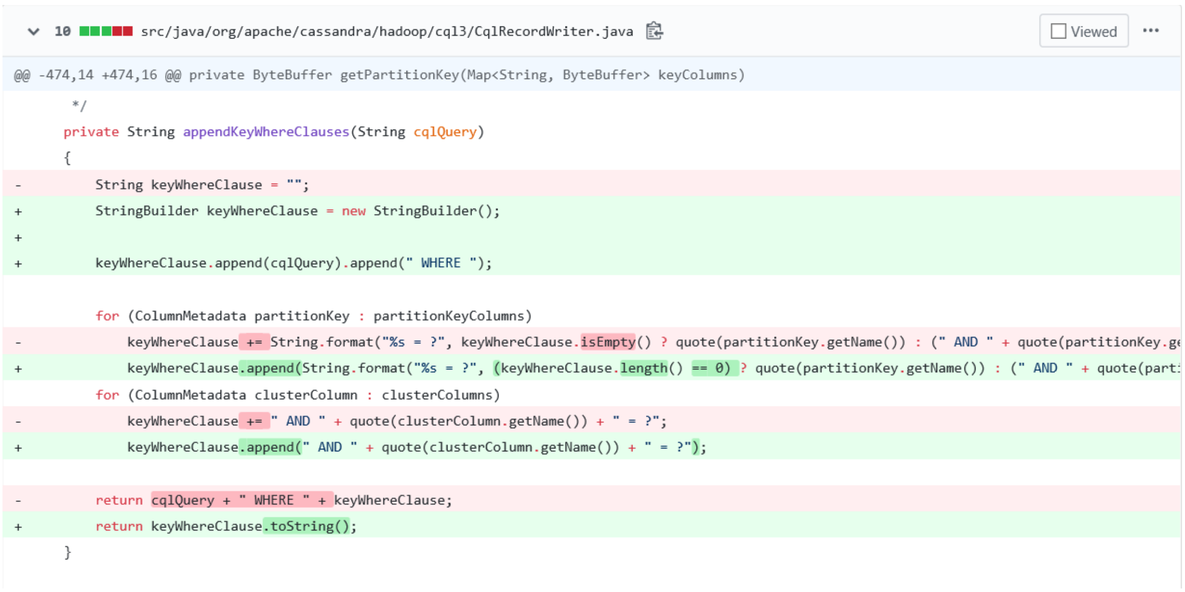
Reviewers
No reviews

Assignees
No one assigned

Labels
None yet

Projects
None yet

We solved the inefficiency caused by string concatenation in loops using StringBuilder.



```
10 src/java/org/apache/cassandra/hadoop/cql3/CqlRecordWriter.java
Viewed
@@ -474,14 +474,16 @@ private ByteBuffer getPartitionKey(Map<String, ByteBuffer> keyColumns)
      */
     private String appendKeyWhereClauses(String cqlQuery)
     {
-        String keyWhereClause = "";
+        StringBuilder keyWhereClause = new StringBuilder();
+
+        keyWhereClause.append(cqlQuery).append(" WHERE ");

         for (ColumnMetadata partitionKey : partitionKeyColumns)
-            keyWhereClause += String.format("%s = ?", keyWhereClause.isEmpty() ? quote(partitionKey.getName()) : (" AND " + quote(partitionKey.ge
+            keyWhereClause.append(String.format("%s = ?", (keyWhereClause.length() == 0) ? quote(partitionKey.getName()) : (" AND " + quote(parti
         for (ColumnMetadata clusterColumn : clusterColumns)
-            keyWhereClause += " AND " + quote(clusterColumn.getName()) + " = ?";
+            keyWhereClause.append(" AND " + quote(clusterColumn.getName()) + " = ?");

-        return cqlQuery + " WHERE " + keyWhereClause;
+        return keyWhereClause.toString();
     }
```