Improve configuration framework [HBASE-13936]

Motivation

// TODO

Current System

- Uses *Configuration* class from Hadoop project to manage configurations. Getter and setters used to access values - *get*("property.name.as.string", defaultValue)* and *set*("property.name.as.string", newValue)* where * can be *Int, Boolean, String*, etc.
- *HBaseConfiguration* class: Loads configurations from *hbase-defaults.xml* (present in *hbase-common*) and *hbase-site.xml*. Returns *Configuration* object.
- *Configuration* object is passed around, copies are made and values are changed using set*() function.
- *ConfigurationManager* class only handles updates to dynamic configurations. Update signal is received via rpc method *RSRpcServices.updateConfiguration*. *ConfigurationManager* maintains a list of objects of classes which implement *ConfigurationObserver* interface. On receiving update signal, *ConfigurationManager* loads the new configuration and calls each observer's overridden method *onConfigurationChange()* which reloads the object's state based on new configuration.
- Some jiras related to dynamic configuration changes: HBASE-3909, HBASE-8302

Issues with current system

1) dynamic configuration documentation is very poor
    a) no list of dynamic configs
    b) shell functionality not well documented
2) Bad design for registering observers
   example from *HMaster*

```
// in function initializeZKBasedSystemTrackers()
this.balancer = LoadBalancerFactory.getLoadBalancer(conf);

// 200 lines later in another function finishActiveMasterInitialization()
```

```
confManager.registerObserver(this.balancer);
```

Why should a class creating *Foo* object care which configs are used by *Foo* internally? So also it should not have to know that Foo needs to be registered as an observer. There are few dynamic configs right now, but as more are added, more objects will need to be registred increasing chance of adding major bugs because of 'forgetting' to register.

Solution: Objects should register themselves with *ConfigurationManager* to get updates.

3) Observers inherit from *ConfigurationObserver* and implement *onConfigurationChange()* to handle updates. This has also led to one-function-catches-all-updates situation which has it's own drawbacks explained below.
    a. Easy to break functionality of an existing dynamic configuration by adding new logic which uses it but forgetting to handle updates to its value.
    b. Tightly couples unrelated configurations

```
onConfigurationChange(conf) {
  x = conf.get(X);
  y = conf.get(Y);
  z = conf.get(Z);
  ....
}
```

In the above example, if updating Y fails and an exception is thrown, updates to Z and other downstream configurations will be skipped.

In class Foo:	In class Bar:

onConfigurationChange(conf) {
  x = conf.get(X);
  y = conf.get(Y);
  z = conf.get(Z);
}
	onConfigurationChange(conf) {
	  z = conf.get(Z);
	  y = conf.get(Y);
	  x = conf.get(X);
	}

Moreover, if a set of dynamic configurations is being updated in multiple places, it is possible that some configurations get updated only partially depending on relative ordering with the failing configuration. For example, in above table if Y fails, X and Z will only be partially updated.

    c. Since failures may throw exceptions, updating multiple configurations and reporting success / failure messages for each configuration using single function is not possible.
    d. Can not support filtering configurations. For e.g. it is not possible to update only *foo.bar, or foo.**

Solution: Update configurations in isolated functions. In such a case, inheritance won't be the right design. More details below.

4. Inadequate shell user experience (commands: *update*config()*)
    a. no progress updates like

i. which machines are pending/in progress/done/failed?
ii. which configurations' updates are pending/in progress/done/failed?

b. no overall success or failure reporting

c. (minor) no command like `*update_config 'X = Y'*` to ephemerally update specific configuration.
(non-persistent configuration changes are generally bad idea because they can easily introduce bugs and are hard to debug since changes vanish on cluster restart, nonetheless, such a function is really useful to SREs in some cases)

Proposed solutions

1) Improve documentation
    a) document list of dynamic configuration, automatically update the list from code.
    b) Update Apache HBase Reference Guide

2) Improve update*config() shell commands: On executing command, show following relevant information updates

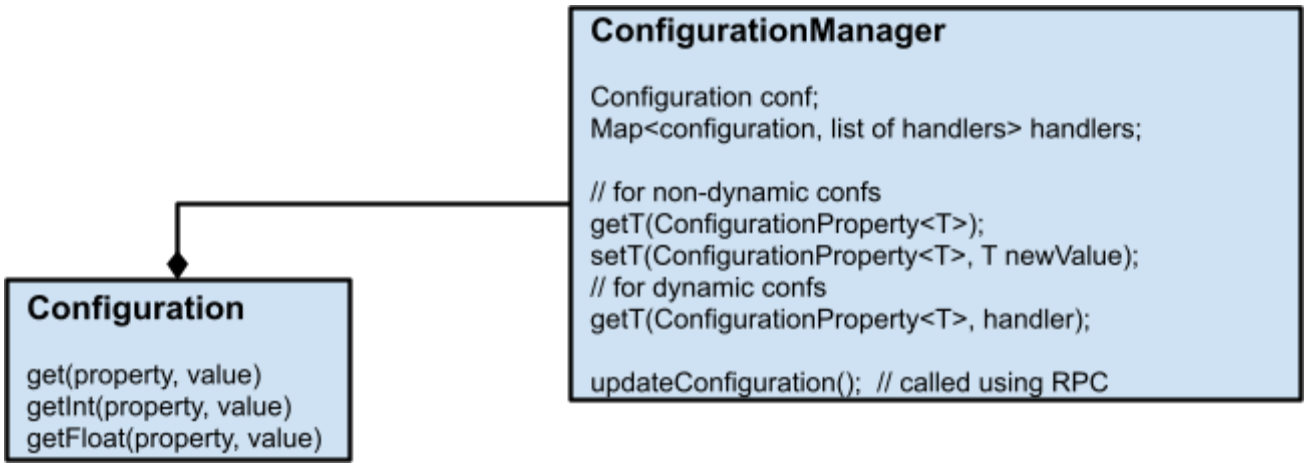
    a) changed configurations
        i) if the configuration is dynamic: old and new value
        ii) if the configuration is not dynamic: show warning that it will be ignored
    b) progress update
        i) by machine: which are pending/in progress/done/failed
        ii) by configuration: which pending/in progress/done/failed
        iii) overall success/failure

3) Code changes
    a) Add *ConfigurationProperty<T>* class which will store all information related to a configuration like name, default value, description, units, etc. It will be single source of truth for any information related to a configuration. Aims to solve following problems in current code:
        i) Different inconsistently named variables for configuration name and default value
        ii) Sometimes, these declarations are spread randomly in the code
        iii) Sometimes, different default values are used from one call of *get*()* to another.
        iv) No type check when setting/getting values. For example, *setInt()* call can be made with a configuration which is supposed to be a boolean and it will succeed thereby corrupting the configuration.

    b) Add new functionality to *ConfigurationManager*.
        i) New getters and setters : *getT(ConfigurationProperty<T>)* and *setT(ConfigurationProperty<T>, newValue)*. No need to specify default value with each call. This will also check at compile time that type of the function and the configuration value are same.
        ii) On configuration change, detect which dynamic configurations changed and report the old and new values to user (to prevent unwanted changes).

iii) On configuration change, detect changes to non-dynamic configuration and report to user that they'll be ignored. This can save the user a lot of trouble from having to figure out why certain changes are not taking effect.

c) Add new function: *get*(property, update_handler)*
To avoid 'forgetting' handling of updates, value of dynamic configurations should be queried via special function which takes in a non-null callback as third argument which will be called if and only if *"foo.bar"* configuration changes.
No changes needed for non-dynamic configurations. However, if get*() (single argument non-handler version) is called with a dynamic configuration, it can return null and/or throw exception.



d) Instead of a single function to update all configuration (*onConfigurationChange()*), design should support updating single configuration/set of configurations in isolation. For example, UpdateFooBar() function below. This will:
   i) allow configuration level success/failure reporting. eg.
   foo.bar successfully updated.
   zing.pooh update failed to update in handler 'winnieThePooh'
   the.lion.king failed to update in handler 'Simba'
   ii) prevent tight coupling with other configuration updates
   iii) [stretch] allow filtering of which configuration to update.

```
class ExampleConfigs {
  public static ConfigurationProperty<Boolean> ABC_XYZ =
    new ConfigurationProperty<>("abc.xyz", true);
  public static ConfigurationProperty<Boolean> FOO_BAR =
    new ConfigurationProperty.Builder<>("foo.bar", false)
        .setIsDynamic().build();
}

class ExampleUse {
  ...
  void someFunction(..) {
    // getting non-dynamic config, no handler argument.
    confManager.getBoolean();
```

```
    ...
    // getting dynamic config
    confManager.getBoolean(, UpdateFooBar);
  }

  class ConfigurationUpdater {
    // updates state related to 'foo.bar' configuration.
    void UpdateFooBar(Configuration conf) {....}
  };
  ...
}
```

If a component of an object's state depends on multiple configurations, single function can be added as the handler for all those configurations. For eg.

```
void updateFromXYZ(conf) {
  // update foo component from X, Y and Z configurations.
}
...
```

Then all X, ..., Z configs will have *updateFromXYZ()* as the handler. ConfigurationManager can de-duplicate references to it and call it only once.

e) Another design change being promoted is 'Keep configurations in the right scope'. For example, client code shouldn't have access to server-only configs and vice versa. Similarly, master code should not have access to regionserver only configs and vice-versa. This means *ConfigurationProperty* declarations should be placed in the right scope.
There will be three main classes - *{Common, Server, Client}Configurations* which will collect *ConfigurationProperty* declarations from code under *hbase-common, hbase-server and hbase-client* respectively using Reflections library. Server and client instances can then use *CommonConfigurations* to get their respective full set of configurations.
So *HBaseConfiguration* will be deprecated but since it is interface public, it will be somehow supported for 1.x releases.

f) Update configuration servlet to show appropriate dynamic configuration information in web UI.

[stretch] When *updateconfig()* is executed, report anomalies in configuration changes detected across servers/RS as warnings. For example. *foo.bar* changed to 10 on one machine and to 100 on all other machines. Probably a misconfiguration?

[stretch, if easily possible] Compile time check to verify right *getConf()* function is called for dynamic and non-dynamic configurations.

Moving from *Configuration* to *ConfigurationManager*

This is no easy task. It'll take few months and efforts from everyone to accomplish. In many cases, the switch won't be easy and will thus required progressive smaller changes. Some of these are highlighted below.:

Case 1:

HConstants contains

```
public static FOO_BAR = “foo.bar”;
public static int FOO_BAR_DEFAULT = ...;
```

Since HConstants is public interface, the right change here can be:

i. Create a new ConfigurationProperty in the right scope

```
class FooConfigs {
  ...
  public static ConfigurationProperty<Integer> FOO_BAR =
    new ConfigurationProperty<>(
        HConstants.FOO_BAR, HConstants.FOO_BAR_DEFAULT);
  ...
}
```

ii. Add deprecated annotations in HConstants.

```
@Deprecated
public static FOO_BAR = “foo.bar”;
@Deprecated
public static int FOO_BAR_DEFAULT = ...;
```

iii. Replace uses of `HConstants.FOO_BAR` in other places with `FooConfigs.FOO_BAR.getName()` and similarly for default value.

Case 2:

```
conf.getType(“parameter”, defaultValue);
```

Changes:

i. Create a new ConfigurationProperty in the right scope

```
class BarConfigs {
  ...
  public static ConfigurationProperty<Type> PARAM =
    new ConfigurationProperty<>(“parameter”, defaultValue);
  ...
}
```

ii. Change function call to

```
conf.getType(BarConfigs.PARAM.getName(),
    BarConfigs.PARAM.getDefaultValue());
```

iii. If possible, change the Class/Function to keep/accept ConfigurationManager as member/parameter instead of Configuration and change the call to `confManager.getType(BarConfigs.PARAM);`

Case 3:

```
conf.getType(“parameter”, value1)
conf.getType(“parameter”, value2)
```

Multiple get*() of same property with different default values.

This is one reason why current design is bad. Multiple defaults are confusing. It's very possible to have a case where no single value works as default for the property. In such a situation, a better alternative might be to have a sentinel value to denote that property is not explicitly set, and then choose the right value depending on situation. Something like:

```
// -1 as sentinal
… FOO_BAR = new ConfigurationProperty<>("foo.bar", -1);
if (confManager.getType(SomeConfigs.FOO_BAR) ==
    SomeConfigs.FOO_BAR.getDefaultValue()) {
  // set to other values on case by case basis
}
```

Drawback: Choosing sentinel value. Empty string or -ve values should work most of the time.

TODO : add other cases when encountered.

When most/all of the uses have been moved to *ConfigurationProperty*, class/functions can be changed to keep/accept *ConfigurationManager* object instead of *Configuration*.

Testing

TODO: how can this framework be tested.
some basic ideas:
- add check: handler registered for only dynamic config X should not call getConf(Y) where Y is another dynamic config.
- no duplicate names in *ConfigurationProperty*

Rough plan of attack

1) HBASE-13957
- Add *ConfigurationProperty* and new functions to *ConfigurationManager*.
- no changes related to dynamic configs will be made
- only few configurations will be moved to new design initially

2) Add components of new dynamic config framework: get*(conf, handler), Map<conf, handlers>, etc and use this new functionality in classes changed above in step 1.
Move ConfigurationManager to hbase-common.

3) Move all classes which are using dynamic configs to new framework

4) Delete old dynamic configs framework: *ConfigurationObserver*

5) Improve shell commands reporting. Maybe add new command "*update_config 'X = Y*'"

6) Testing framework for dynamic configs

7) Misc: automate *ConfigurationProperty* → Ref Guide configuration documentation (when all configs have been added as *ConfigurationProperty*, we should be able to delete hbase-default.xml).

8) Document dynamic configs framework

Open questions

1) Updates will happen one configuration at a time (iterate over registered handlers) instead of one component at a time (*onConfigurationChange()* updating all configs together). Can this lead to any issues?

2) Configuration changes which are not backward compatible should be avoided. For example, when a region moves from an updated RS to a non-updated RS, it can cause troubles. More thoughts?

Meeting on 20150625

Look for Reconfigurable in Hadoop. A new Hadoop 'framework'.

Talk to Darren in CM team on what ideal config would be.

In the config., actually state what max and min are, etc…. allowable values. A good one is making sure they enter milliseconds rather than seconds and so on.

How to do ENUMS in Configuration?

HBASE-13936 Configuration

Tooling.

Can we simplify? Matteo points out YAML or java property files is not right direction.. does not solve the problem.

Come up w/ design and pattern, do current dynamics, add a few more than do piecemeal.

Configuration scoping by package.

Survey of other projects doing configuration.

Find for appy offline the nicolas configuration stuff.

Can we go even simpler

Config, Rule check, and Action.

Reflection to figure the method to call.. .but have to pass a String…. for method name.

Can't have separate config class because 'actions' want to mess with internals.

Startup crash the server but when dynamic config we want to reject.

At runtime, can see what is bad….. it is easy in UI.

Tool to check.

So, add dry-run of suggested configs. Check basic stuff. Flag configurations as resources.

Numeric and string.

Resource category, which is server or process level.

An example. Instead.

Who is responsible for parse, type check?

Ignore stuff below

- partition configs into master/rs/both: for users to better understanding effects of dynamically changing configs, and generally good too
- choose top 20 configs to make dynamic
- working: script (shell?) sends signal to all masters/RS (serial/parallel?) (exits/waits to report?). See how it's done today.

- inner class ConfigurationUpdater to keep all functions which'll handle conf updated. will keep pointer to outer class
- outer class: add initialize() method which will first instantiate ConfigurationUpdater and then other components depended on dynamic confgs. Handlers will be CU's function