Other languages: 繁體中文

Plugins

Plugins are use to help users build complex applications. In Drogon, all plugins are built and installed into the application based on the configuration file. Plugins in Drogon are single-instance, and users can implement any functionality they want with plugins.

When Drogon runs the run() interface, it instantiates each plugin one by one according to the configuration file and calls the `initAndStart()` interface of them.

Configuration

Plugin configuration is done through the configuration file, for example:

```
    "plugins": [
        {
        //name: The class name of the plugin
        "name": "DataDictionary",
        //dependencies: Plugins that the plugin depends on. It can be
commented out
        "dependencies": [],
        //config: The configuration of the plugin. This json object is the
parameter to initialize the plugin.
        //It can be commented out
        "config": {
        }
        }],
```

It can be seen that there are three configurations for each plugin:

- name: is the class name of the plugin (including the namespace). The framework will create a plugin instance based on the class name. If the item is commented out, the plugin becomes disabled.
- dependencies: Is a list of names of other plugins that the plugin depends on. The framework creates and initializes all plugins in a specific order. Prioritize the creation and initialization of plugins that are dependent by others. At the end of the program, plugins are closed and destroyed in reverse order. Please note that circular dependencies in plugins are forbidden. Drogon will report an error and exit the program if it detects a circular dependency. If the item is commented out, the list of dependencies is empty.
- config: is the json object used to initialize the plugin, the object is passed as an input parameter to the plugin's `initAndStart()` interface. If the item is commented out, the json object passed to the `initAndStart` interface is an empty object;

Definition

User-defined plugins must inherit from the drogon::Plugin class template, and the template parameter is the plugin type, such as the following definition:

```
class DataDictionary : public drogon::Plugin<DataDictionary>
{
public:
    virtual void initAndStart(const Json::Value &config) override;
    virtual void shutdown() override;
    ...
};
```

One can create source files of plugin by drogon_ctl command:

```
drogon_ctl create plugin <[namespace::]class_name>
```

Getting Instance

The plugin instance is created by drogon, and the user can get the plugin instance through the following interface of drogon:

```
template <typename T> T *getPlugin();
```

Or

```
PluginBase *getPlugin(const std::string &name);
```

Obviously, the first method is more convenient. For example, the DataDictionary plugin mentioned above can be obtained like this:

```
auto *pluginPtr=app().getPlugin<DataDictionary>();
```

Note that it is best to get the plugin after calling the framework's run() interface, otherwise one will get an uninitialized plugin instance (this doesn't necessarily lead to an error, it is ok to just make sure to use the plugin after initialization). Of course, since the plugin is initialized in a dependency order, it is no problem to get the instance of another plugin in the `initAndStart()` interface.

Life Cycle

All plugins are initialized in the run() interface of the framework and are destroyed when the application exits. Therefore, the plugin's lifecycle is almost identical to the application, which is why the getPlugin() interface does not need to return a smart pointer.

Next: Configuration File