Data Manager 2

Структура и интерфейсы

Николай Шадрин

6 апреля 2014 г.

Аннотация

Документ описывает представление данных в Data Manager 2, и объясняет, почему оно именно такое. Документ определяет интерфейсы к Data Manager 2 и механизмы расширения Data Manager 2 для реализации конкретных задач.

1. Что такое Data Manager 2?

DM2 - это набор классов, реализующих обмен данными между разными частями системы UCS, а именно:

- компонентами внутри основного процесса UCS;
- программами на UCSL;
- внешними процессами UCS;
- удаленными компонентами UCS, такими, как терминал.

К реализации DM2 предъявляются следующие требования:

- доступ к значениям переменных должен быть эффективным;
- данные должны быть структурированы;
- реализация должна быть легко расширяемой и модифицируемой под конкретную задачу.

DM2 не является "хранилищем переменных"; кроме того, DM2 не определяет сетевой протокол в случае обмена данными по сети. Более того, DM2 не определяет конкретные типы данных, которыми будут обмениваться компоненты; однако, конкретная реализация DM2, разумеется, требует спецификации типов.

2. Представление данных в DM2

DM2 определяет концепцию "класса-значения" (UCSValue) и "класса-интерфейса к значению" (например, UCSInt). Класс-значение содержит, собственно, значение переменной; в случае примитивного типа, например, *int*, он просто содержит поле типа *int*. Конкретный экземпляр "интерфейса к значению" определяет способ доступа к значению - например, он может содержать указатель на это значение.

Для получения доступа к значению клиент DM2 создает экземпляр класса-интерфейса к значению требуемого типа (например, экземпляр UCSInt), который связывается с определенным экземпляром класса-значения (например, UCSPrimitiveValue<int>). Другой клиент DM2 может создать свой экземпляр UCSInt,который тоже будет связан с этим значением; таким образом, обеспечивается общий доступ к данным.

Для того, чтобы два клиента DM2 могли получить доступ к одному и тому же значению, должен быть какой-то механизм именования значений. Для этого DM2 определяет вводит концепцию *контейнера* - значения, содержащего другие значения.

Определены следующие типы контейнеров:

UCSNamespace
: содержит неопределенное число именованных значений; ключ - строка (*std::string*). Значения могут быть произвольных типов.

UCSVectorValue<T>
: содержит неопределенное число значений типа T, индексируемых целочисленным неотрицательным индексом.

Необходимо обратить внимание, что контейнеры содержат *экземпляры классов-значений* - например, UCSPrimitiveValue<int>, а не примитивы (например, int). Поскольку контейнеры сами являются значениями (UCSNamespace и UCSArray наследуются от UCSValue), то контейнер может содержать контейнер. Конструкции типа "массив структур" и "структура, содержащая массив" должны работать корректно.

Каждому классу-значению (унаследованному от UCSValue) соответствует класс-интерфейс, в частности:

Класс-значение	Класс-интерфейс
UCSNamespace	UCSStruct
UCSPrimitiveValue<T>	UCSPrimitive<T>
UCSVectorValue<T>	UCSVector<T>

2.1. Требования к классу-значению

Класс-значение должен:

- быть унаследован от UCSValue;
- иметь default constructor;
- иметь copy constructor;
- если значение должно передаваться по сети, то класс-значение должен быть унаследован от UCSSerializable и реализовывать соответствующие методы.

Контейнеры, хотя и являются значениями, не передаются по сети (передаются только элементы). См. далее "Сериализация".

2.2. Требования к классу-интерфейсу

Класс-интерфейс должен:

- иметь default constructor, создающий пустое значение (используя default constructor соответствующего типа-значения);
- иметь конструктор (shared_ptr<UCSValue>), создающий интерфейс к данному значению;
- иметь метод getUCSValue(), возвращающий shared_ptr указатель на значение;
- иметь поле typedef X valueType, где X - тип класса-значения.

2.3. Цикл жизни значений

Значение существует, пока есть хотя бы один интерфейс к значению. Поэтому класс-значение содержит shared_ptr указатель на класс-значение.

Класс-контейнер содержит только *weak_ptr* указатели на значения, так как, например, существование переменной в UCSNamespace актуально только до тех пор, пока есть клиенты, адресующие это поле. Для того, чтобы контейнер мог удалить из себя значение, которое больше никто не использует, класс UCSValue определяет событие onDelete, вызываемое в деструкторе UCSValue; класс UCSNamespace регистрирует себя как получатель этого события при добавлении в себя значения.

3. Связывание интерфейсов и значений

Для того, чтобы клиент DM мог использовать какие-то общие данные, он должен создать экземпляр класса-интерфейса (например, UCSInt), передав ему указатель на соответствующее значение (например, UCSPrimitiveValue<int>). Чтобы как-то получить этот указатель, требуемое значение должно содержаться в контейнере - например, в UCSNamespace, откуда клиент может получить этот указатель по имени значения, или в UCSVectorValue, откуда его можно получить по индексу. Это означает, что весь набор общих данных должен содержаться в каком-то контейнере верхнего уровня; логично, что этот контейнер - UCSNamespace, т.к. удобно адресовать значения по имени. С практической точки зрения это означает, что все клиенты DM получают указатель на корневой контейнер, и создают свои общие данные в этом контейнере.

Фактически, это означает, что указатель на корневой контейнер - это и есть интерфейс к DM с точки зрения клиентов.

3.1. Метод UCSNamespace::get

Метод UCSNamespace::get<T> (const string& key) возвращает указатель (shared_ptr) на именованное значение в UCSNamespace. Если значения с таким именем нет, оно будет создано; при этом параметр T - это класс-интерфейс к значению нужного типа, а UCSNamespace создаст объект типа T::valueType - этим обьясняется необходимость существования поля valueType в классе-интерфейсе.

Почему так?

- тип значения как-то нужно задать в любом случае;
- указывать тип значения непосредственно неудобно, т.к. клиент не имеет дело со значениями напрямую; в то же время, тип интерфейса клиенту известен.

Практически это означает, что клиент может делать:

```
UCSInt x (rootNamespace –> get<UCSInt> ("mySuperInt");
x = 5;
```

3.2. Поля структур

Структура - это интерфейс к UCSNamespace. Практически, это означает, что класс UCSStruct содержит указатель на UCSNamespace; этот указатель называется fields.

Клиент DM может определить структуру с каким-то набором полей, создав класс, унаследованный от UCSStruct, следующим образом:

```
class UCSTestStruct: public UCSStruct {

      public:

            UCSInt field1 = UCSInt (fields –> get<UCSInt>
               ("superField"));
            UCSInt field2 = UCSInt (fields –> get<UCSInt>
               ("anotherSuperField"));

      public:

            UCSTestStruct (): UCSStruct () { }
            UCSTestStruct (shared_ptr<UCSValue> value):
```

```
                UCSStruct (value) { }
};
```

Затем, клиент может создать экземпляр этой структуры, привязанный к значению в rootNamespace:

```
UCSTestStruct testStruct (rootNamespace -> get<UCSTestStruct>
   ("mySuperStruct");
testStruct.field1 = 12345;
testStruct.field2 = 88888;
```

Возможна ситуация, когда разные клиенты привязывают разные интерфейсы к одному и тому же значению-контейнеру. Например, клиент 1 может создать структуру с полями "field1" и "field2", а клиент2 - с полями "field2" и "field3" и привязать их к значению "mySuperStruct" в rootNamespace. Это нормальная ситуация, которая будет корректно обработана. Практически это может возникнуть в следующих случаях:

клиенту нужна только часть полей:
остальные поля просто не будут связаны;

клиент не знает о существовании других полей:
например, в ходе разработки были добавлены новые поля. Клиенты, которые используют только старые поля, будут работать корректно.

Поскольку UCSNamespace является значением, то структуры имеют такой же цикл жизни, как и примитивы, т.е. пока существует хотя бы один интерфейс, существует и структура. Ситуация, когда структура (т.е. UCSNamespace) удалена раньше, чем удалены поля, является нормальной - связанные поля останутся связанными; однако, новые клиенты очевидным образом не смогут связаться с полями удаленной структуры.

3.3. Проверка типов

При связывании интерфейса со значением производится проверка типа (при помощи dynamic_pointer_cast). Если тип значения не соответствует привязываемому интерфейсу, произойдет exception.

4. Функции

Специальным типом значений являются вызываемые функции. Функции могут быть вызваны:

- из нативного кода;
- из UCSL-кода;
- удаленно.

Поэтому, как следствие, к абстрактной функции в общем случае предьявляются следующие требования:

- функция должна принимать на вход аргументы-значения (т.е. vector<shared_ptr<UCSValue>>);

- функция должна уметь выполняться асинхронно, поэтому возвращаемым типом является future<shared_ptr<UCSValue>>.

Функции, как и любые другие значения, могут быть добавлены в контейнер. Собственно, для того, чтобы функцию могли использовать несколько клиентов, она должна быть добавлена в какой-нибудь контейнер - например, rootNamespace. Разумеется, функция может быть полем структуры.

Значением функции является ее реализация, а интерфейсом - объект для вызова функции.

4.1. Асинхронное выполнение - почему future?

Реализация функции не знает, как именно будет ждать ее выполнения тот, кто ее вызвал. Это может быть RPC-клиент, нативный тред, или же выражение в программе на UCSL. Наиболее общий механизм - это callback, но через callback неудобно передавать exception, поэтому DM2 использует future, как стандартный механизм, предлагаемый STL для асинхронных вызовов.

Реализация не обязана быть асинхронной - если функция выполняется *очень быстро*, то она имеет полное право вернуть future с уже готовым результатом.

4.2. Реализация функций

Наиболее общий способ реализации - создание подкласса UCSFunction, реализующего метод execute:

```
class UCSFunction: public UCSValue {

            private:

                        // ....

                        virtual future<shared_ptr<UCSValue>> execute
                            (const vector<shared_ptr<UCSValue>>&
                            params) = 0;

};
```

Это крайне неудобно для нативного кода, поэтому DM2 содержит две обертки для нативных функций:

UCSNativeFunction для асинхронно выполняющихся функций;

UCSNativeBlockingFunction для асинхронно выполняющихся функций;

UCSNativeFunction представляет как UCSFunction функцию, передаваемую в виде std::function<R(A...)>, где R - future<shared_ptr<UCSValue>. Нативная реализация передается непосредственно в конструктор UCSNativeFunction и может быть лямбда-выражением или указателем на функцию или метод.

UCSNativeBlockingFunction представляет как UCSFunction функцию, передаваемую в виде std::function<R(A...)>, где R - класс-интерфейс к UCSValue. Например:

```
     shared_ptr<UCSFunction> testFuncPtr (new
            UCSNativeBlockingFunction<UCSInt (UCSInt,UCSString)> ([]
               (UCSInt x, UCSString str) -> UCSInt {
                  cout << "in testfunc" << endl;
                  cout << "x: " << x << endl;
                  cout << "str: " << (string) str << endl;
```

```
            return UCSInt (x+1);
        }));
```

Эти обертки реализуют метод execute(), который разбирает переданный ему вектор из UCSValue, подставляя эти параметры в std::function, и вызывает этот std::function.

4.3. Вызов функций

В общем случае, для вызова функции необходимо создать массив из ее параметров в виде вектора UCSValue, затем вызвать метод execute() и получить результат через future. Это неудобно для нативного кода, поэтому DM2 содержит обертку UCSNativeFunctionCall, позволяющую передавать параметры просто как параметры и получать результат как результат, например:

```
UCSNativeFunctionCall<UCSInt (UCSInt,UCSString)>
   testFunctionCall (testFuncPtr);
UCSInt result = testFunctionCall.call (6, string ("oh my
   caller"));
```

При этом UCSNativeFunctionCall создаст вектор из UCSValue, заполнит его параметрами вызова call() и вызовет метод execute() реализации.

Таким образом, используя этот механизм, программа на UCSL может вызывать функции в нативном коде и наоборот; кроме того, становится возможен удаленный вызов функции - конечно, в случае, если ее параметры могут быть сериализованы.

5. Удаленный доступ

DM2 включает механизм предоставления удаленному клиенту доступа к общим данным. Под *удаленным клиентом* здесь имеется в виду клиент, который не может непосредственно адресовать значения по указателю; он не обязательно должен быть удаленным физически или выполняться в другом процессе.

Для получения доступа к удаленным данным клиент DM создает объект-прокси (UCSRemotingClient), передавая ему указатель на корневой контейнер (UCSNamespace). UCSRemotingClient использует какой-то абстрактный транспорт для передачи сообщений в соответствующий объект UCSRemotingServer на сервере; механизм передачи сообщений может быть произвольным. В рамках одного процесса сообщения могут передаваться непосредственным вызовом функции.

5.1. Связывание удаленных значений

При своем создании объект UCSRemotingClient регистрирует себя как обработчик события onValueAdded в корневом контейнере. После этого, когда в корневой контейнер добавляется значение, UCSRemotingClient присваивает этому значению определенный уникальный id, и передает сообщение "значению с ключом таким-то сопоставлен id такой-то" на сервер. Получив это сообщение, UCSRemotingServer создает объект UCSValueProxy, который привязывается к соответствующему значению на сервере и регистрирует себя как обработчик события onChange. При изменении значения UCSValueProxy посылает сообщение "значение с таким-то id изменилось" в UCSRemotingClient, который его обрабатывает и изменяет локальное значение; передача значения в обратную сторону происходит аналогично.

Таким образом, работа с удаленным DM выглядит для клиента точно так же, как и работа с локальным.

5.2. Блоки апдейтов

Очевидно, что в случае физической сети передавать датаграмму на каждое изменение каждого значения - непозволительная роскошь. Поэтому при обработке onChange() соответствующий объект UCSValue помещается в очередь апдейтов, если его там еще нет.

Когда транспорт сигнализирует о готовности передать очередную датаграмму, текущий блок апдейтов фиксируется и передается транспорту для сериализации и дальнейшей передачи.

5.3. Точки синхронизации

Любое событие, не являющееся апдейтом значения - например, вызов функции, сообщение о связывании или удалении значения - является точкой синхронизации, при которой текущий блок апдейтов фиксируется и передается транспорту как готовый к передаче; затем передается сообщение, вызвавшее синхронизацию, после чего начинает формироваться новый блок апдейтов. Таким образом, обеспечивается корректность состояния в промежуточных точках.

6. Текущее состояние реализации

На текущий момент не реализованы следующие компоненты DM2:

- абстрактный интерфейс к транспорту;
- сериализация существующих типов данных;
- связывание функций (вызовы функций реализованы).