

设计模式浅谈

coolbacon 硬件十万个为什么 Yesterday

最近发了一些设计模式的文章。RTEMS的内核是面向对象设计的，里面免不了会有设计模式。RTEMS内核是C的，所以，看起来，和OO语言设计的是有出入的，不过思路上是无差异的。我学习设计模式很久了，也尝试用设计模式在自己的项目中，发现总是画虎不成反类犬，体会不到设计模式的威力。

我努力尝试着如何应用设计模式，发现想用好并不是那么容易的事情，而且，搞不好就是过设计或者是欠设计。这些促使自己思考真正主导设计模式的是什么呢？OO方法论的提出，是对过程方法的升级和提高，目的在于提高解决问题的效率。说得更直白些，就是复用代码，减少重复的工作。

代码的复用，实际上是在寻找项目与项目之间的共同的地方；或者换一个说法，就是想办法对抗系统的变化。对抗的目的是，让项目自身发展的过程中，或者一个项目和另外一个项目之间：1.共同的部分尽可能的多（要兼顾 代价与收益的平衡）；2.不同是必然的，减小这种

不同带来的冲击与修改。谈到这里，这个目标如何做到呢？我认为，市场决定产品的功能，产品的功能选择用什么样的技术支撑。市场抽象出来，实际上是一条条需求。这些需求会发生变化。设计好一个系统，也就是对需求变化部分的预测与判断，然后用适当的方法对抗这些设计的变化。说到这里，用好设计模式，实际上是要对自己系统深入了解，并对系统未来的变化进行预测，对抗修改的过程。

设想一下，如果只是简单的设计一个全新的系统，而这个系统你知道只是一个项目。做完拉倒，需求也不会发生变化。我的做法是，怎么简单怎么来，大部分情况下是不会考虑用设计模式的，也不会考虑用什么好的框架。如果用了，也是从别的项目中拷贝过来的，绝不会为其专门考虑的。道理很简单，因为我不需要对抗未来的变化，需求简单的实现就好了。同样的，如果我需要做一个类似楼宇自动化的控制系统，这个系统未来还有多个产品系列，高中低档产品有不同的层次，解决的问题也呈现多样性、延续性和复杂性。那么，如果为了将来少点996，必要的需求预测，分析未来需求变化，选择合适的设计模式，那是一定少不了的过程。

来，举个例子

假设，我们一个家用的门锁，一个较为简单的处理器(Cortex-M3)，需要存储数据，我们要设计一个非易失性数据的存储系统。怎么设计呢？首先，要来看看我们的需求。

- 1.家用的门锁，有用密码打开，那么需要存储密码；
- 2.有指纹信息，需要存储指纹信息；
- 3.有开门方式，需要存储开门方式；
- 4.有一些其他的设置数据，比如说，蓝牙开锁，要不要和网关通讯及通讯的必要设置等等。

简单的列了一下这几条需求。这是不够的，比如说：

- 1.数据存储在哪里？
- 2.固件升级的时候，数据不能丢失，版本之间的数据能向下兼容，最好版本之间的数据完全兼容，向上向下升级，都不会影响系统正常工作；
- 3.要不要断电（异常死机）的时候，不丢失数据？
- 4.数据错乱的时候，系统要能正常工作，不能宕机。

我们分析这个需求。从需求上看，密码、指纹、开门方式等这些数据，有可能会因为产品的升级而发展。也就是说我们首要对抗的修改是数据未来的变化。变化有：

- 1.增加或者删除数据项目；
- 2.改变存储的介质；

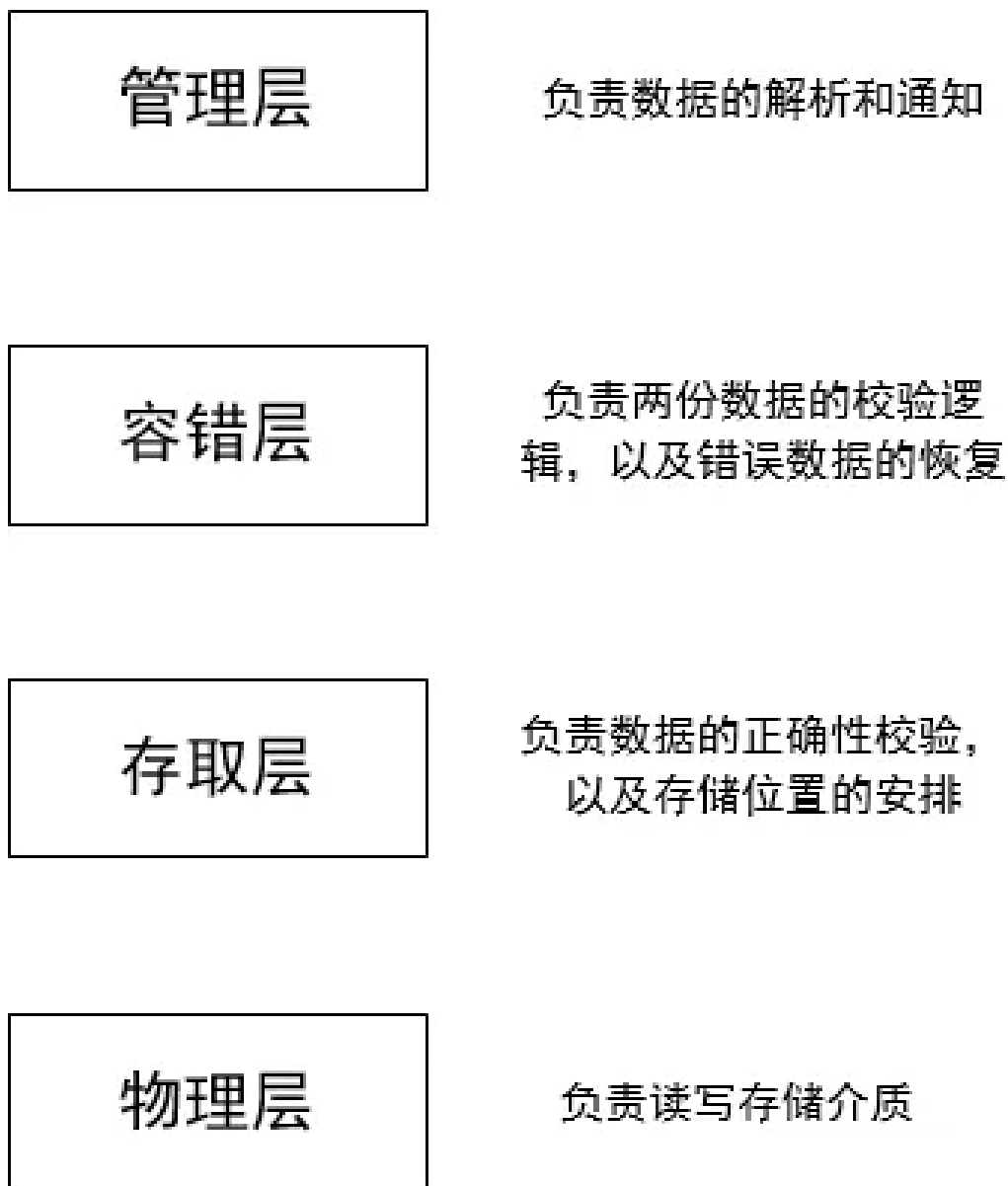
3.数据本身会发生变化，如格式，长度等。

除此之外，也会有如下问题：

- 1.这些存储数据应该是全局唯一的；
 - 2.使用到这些数据的地方分布在代码的各个地方，数据改变时，有可能要通知到这些地方。
- 至于数据错乱，系统中应该固化一组能用的数据。必要时，采用固化的数据工作。

断电(异常死机)保证数据不坏，主要的思路还是软件思路。在物理存储系统上保存两份数据，同一时间只操作一份数据。这样，如果操作成功，自然没问题。如果操作失败，至少有一份数据是好的。通过比对数据的版本和校验，来决定用哪一组数据。坏的数据，在上电的过程中，用好的数据修复。这样就能抵抗大多数不确定的场景。

看来，一个简单的数据存储，就被这样预测着，分析着，搞得越来越复杂。不过，都是实际需要。这么多功能，怎么组织呢？这里我们自始至终关心的是数据的流向，我们是围绕数据加工的一个流程。存储介质是仓库，类似的进仓库出仓库的一个模型。参考OSI的七层理论，我们也可以设计出存储的层理论：



RT-Thread

这个物理层，一般就是我们说得驱动。该层要解决的问题主要是读写存储介质。既可以操作实际的物理芯片，如果嵌入式系统足够的大，支持文件系统。也可以在该层修改成文件系统。

存取层，所有适当大小的数据，需要校验，以验证数据的好坏，还负责安排数据的具体存储位置。但对于存取层来说，它不关心具体的数据内容是啥，只关心该块的数据正确不正确。

容错层，负责两份数据的校验逻辑，以及一份数据恢复另外一份数据的逻辑。

管理层，主要负责具体的数据解析，以及没有正确数据，用缺省数据提供给其他系统。还有，更改了数据，及时的通知其他用到数据的地方。


这么一划分，是不是更清晰了一点呢？我们一层层看。

物理层这里怎么设计？要对抗什么变化呢？物理层主要对抗的未来变化，就是各种存储介质的切换。如果用设计模式，那么毫无疑问，采用工厂模式。

```
1 int read(uint32_t address, void *data, uint32_t size);  
2 int write(uint32_t address, void *data, uint32_t size);
```

这两个是主要的函数。嵌入式里资源受限，一般项目都是用纯C写成，我们可以仿照C++的一些实现思路获得设计。不同的存储介质都需要提供相应的函数。

```
1 typedef struct tagRwPhyFunc {
2     int (*init)(void);
3     int (*read)(uint32_t address, void *data, uint32_t size);
4     int (*write)(uint32_t address, void *data, uint32_t size);
5     int (*getInfo)(uint32_t *erase_size, uint32_t *write_size, uint32_t *size);
6     int (*erase)(uint32_t address, uint32_t size);
7     int (*eraseAll)();
8     int (*close)(void);
9 } TRwPhyFunc;
10
```

 RTMS

通过工厂函数获取到相关的接口。以下即是不严谨的实现，看个意思，领会精神。不建议采用模仿C++的写法，丢失了C语言的灵魂，简洁有效，是我们的第一目的。C扩展起来也很简单，增加一个设备类型，驱动接口，然后增加工厂模式里的else if分支即可完成新的物理层的添加。其他都无需改变。

```

1  int getRwPhyFunc(TRwPhyFunc *funcs, int type)
2  {
3      if (type == NV_FLASH)
4      {
5          funcs->init = flash_init;
6          funcs->read = flash_read;
7          funcs->write = flash_write;
8          funcs->getInfo = flash_getInfo;
9          funcs->erase = flash_erase;
10         funcs->eraseAll = flash_eraseAll;
11         funcs->close = flash_close;
12     }
13     else if (type == NV_FILESYSTEM)
14     {
15         //...
16     }
17     else
18     {
19         return FAILED;
20     }
21     return SUCCESS;
22 }
23

```

 RTEMS

存取层需要对抗什么问题呢？如果自己操作非易失性存储设备，要考虑擦写的问题。擦写的大小和写入的大小并不一致。这就有可能要开辟一个比较大的缓冲区，一次性读入，然后再写入。上层过来的数据可能会比一次性写入的大，或者比一次性写入的小。小的话，直接分配一次性的读写页面。大的话，可能要分开了。另外一个问题，就是逻辑的地址到实际地址的转换。这层问题就这么多，为了有效的管理这些策略，可以考虑用一

个 **策略模式**。如果不想用，直接上，我想问题也不是太大。主要看系统的承载能力，这里设计上就有一定的弹性。没有必要的话，就以简单原则为第一原则。

这里还有一个比较头大的问题。即过大的数据块分成小数据块的存储问题。按理说，这个问题是由存取层去控制，上层压根就不知道这事情。要么呢，从设计上规避这个问题，限制一下，一个数据块最大，比如说64字节；如以太网数据包一样，由上层去组包去。要么本层想办法解决，解决的要点是获取数据的大小，要上层一次性提供所有数据的大小给本层（因为要安排数据存储位置），本层负责数据的封包组装。但我不认为这个是个好主意。主要应用层添加一些数据项后，数据变大了，波及的层就多了。如果采取固定最大尺寸的方式，那么实际上我们可以定义一套数据存储的协议。常见的Flash的页大小，有128, 256字节等。我们可以选取128字节；

存取层需要校验，采用什么算法呢？比如说crc32,那么需要额外的4个字节存储校验。那么就剩下124字节了。容错层可能要存储数据的版本信息，用于两份数据的PK，可能又需要4个字节。

容错层 同存取层，顶多可以考虑一个 **策略模式**，没有更多的内容了。

管理层，那是设计的重点。首先，全局只有一个，那么毫无疑问，要使用**单例模式**。其次，要通知到对应的数据使用者，那么还要用一个**观察者模式**。这两个模式主要是解决功能的问题多一些。观察者模式有利于解决数据添加后，需要添加新的通知模块。这个也可以对抗这部分修改。我们的大头还是来自数据本身的设计。

C 语言设计的话，主要的思路还是使用静态表。即上面我们定义了最大的一个数据包也就120个字节。那么：

我们定义一些要存储的数据结构，比如说：

```

1  typedef struct tagBleCfgData {
2      //...
3  } TBleCfgData;
4
5  typedef struct tagFingerPrintData {
6      //...
7  } TFpData;
8
9  typedef struct tagOtherData {
10     //...
11 } TOtherData;
12
13 typedef struct tagDataItem {
14     uint32_t index;
15     uint32_t version;
16     uint32_t size;
17     void *data;
18 } TDataItem;
19
20 const TDataItem gDataTbl[] = {
21     {1, VERSION_1_0, sizeof(TBleCfgData), &bleCfgData},
22     {2, VERSION_1_0, sizeof(TFingerPrintData), &fpData},
23     {3, VERSION_1_2, sizeof(TOtherData), &otherData}
24 };


```

RTEMS

那么，读取数据就围绕这个表展开了。bleCfgData就是全局的存储蓝牙配置数据的数据结构体。读取数据的时候，为了方便，可以采用index编号作为索引。可以预定义一个枚举或者宏，让代码的可读性更高。在C语言里，只有数据有单例模式；函数天生就是单例的。另外讲到观察者模式，观察者模式无非就是要维护一个通知列表，这个列表可以是动态的，也可以是静态的，消耗的内存不一样。对于代码中确定的行为，多半都是静态的。这里呢，有几个办法，一个是，做一个总的update的函数；添加时就在这个总得update里添加就好。比如

说以上的数据表改为：

```
1 typedef struct tagDataItem {
2     uint32_t index;
3     uint32_t version;
4     uint32_t size;
5     void *data;
6     void (*update) (void);
7 } TDataItem;
8
9 const TDataItem gDataTbl[] = {
10     {1, VERSION_1_0,      sizeof(TBleCfgData),  &bleCfgData,      bleCfgUpdate},
11     {2, VERSION_1_0,      sizeof(TFingerprintData),  &fpData,      fpDataUpdate},
12     {3, VERSION_1_2,      sizeof(TOtherData),  &otherData,  otherDataUpdate},
13 };
14
15
16 void bleCfgUpdate(void)
17 {
18     updateFunc1();
19     updateFunc2();
20     ....
21 }
22
```

 RTEMS

如此这般，其实也是一种简洁有效的观察者模式实现的方式。当然，还有更高级的玩法：

修改连接脚本。为蓝牙的回调函数建立一个单独的段。然后，通过连接指令，将蓝牙通知回调函数全部放入该段。当数据发生变化时，代码调用该段的函数。每4个字节就是一个函数指针。顺次调用就好。

不过，我并不推荐这么玩。因为连接脚本和代码中定义段，将函数放置到特殊段的指令都不是标准的。代码的可移植性比较差，如果是底层技术可以用，高层，尤其是应用层，可以着重考虑我提供的第一个方法。当然也有很多变种，设计的弹性比较大，但思路都是一样的。

以上就是我粗浅的理解，希望抛砖引玉。如果喜欢我的博文，请多多支持我的公众号。谢谢。

Forwarded from Official Account



RTMS >
