概述

“防御性编程”是软件开发中十分有用的思想。人类在数十年的软件开发历史中越来越多地发现，软件开发中的首要问题是如何降低日益增长的软件复杂度。软件的复杂度和其功能要求往往以一对矛盾的形式出现，如何消除或者矛盾，成为软件开发过程中我们不得不思索的问题。一旦软件复杂度可以通过某种机制来减弱，高质量的软件将会更有机会开发成功。本文将展示Map-Walker所采用的机制是如何一步步消除软件开发复杂度的。

选择优秀并且适合的开发语言

针对项目本身选择合适的程序开发语言，是项目开发首先应该考虑的问题之一。近年来，动态语言作为一种新兴的编程语言种类开始逐步兴起。它给开发者带来的便利性和敏捷性是传统编译型语言难以企及的。在本项目中我们毫不犹豫的选择了Python与C++作为主要开发语言，意在充分利用二者的优点，同时可以借助他们来尽可能降低软件开发的复杂度。

Python——优雅神速的利剑

Python是最近兴起的动态语言的领袖。我们选择使用Python作为软件内部逻辑的开发语言。我们选择它的原因有：

1. 快速开发原型的能力。快速开发程序原型，是应用程序开发之道的首要思想。通过快速开发原型，可以在第一时间了解到程序开发的可行性与复杂点、瓶颈所在。并且通过用雏形和用户的接触，软件公司得以明确用户真正需要的功能所在，从而减少人力与财力的浪费。古往今来，有无数的软件开发公司因其不能生产出用户真正需要的产品而导致项目失败。软件的开发者，甚至需求分析人员和真正的客户之间会有难以磨灭的交流隔阂，需求分析人员难以很贴切地理解用户真正的需求。唯一验证软件开发“步入正轨”的方式，就是尽快做出程序的雏形，呈现给用户，让他们分辨这是不是自己所要的。Python作为一种简单而高效的开发语言，极大地便利了软件开发。在程序语法上相比传统的编译型语言具有压倒性的简洁性和灵活性，从而为快速开发原型提供了快车道。
2. 丰富的库和接口。Python的第二个优势是其得天独厚的生态环境。如今，成千上万的黑客已经向Python社区贡献了数以千计的高质量代码，让普通开发者也可以非常便利的获取。我们的项目中，诸如thread，socket， os等开发库是必不可少的。与Python本身简单的语法相得益彰，这些库变得简单易学。
3. 简单易学。通过简单的交流与在线学习，所有组内成员都能在极短的时间内掌握Python的核心语法和编程范式。
4. 速度绝不是问题。动态解释型语言往往会因为代码运行速度而广为人所诟病，但这已经是上世纪的思维。如今计算机每秒的运算速度可以达到几千亿次，这就构成了一种拉平效应，编译型语言和解释型语言在一些对时效要求不是很高的场合表现的差异并不大。因此，尽管Python的运行效率只有C++的二百分之一，但是对于小型项目工程来说已经足够用。

Python是我们强大的武器，对于我们小组来说，它就是一颗“银弹”。

C++——快速，交互

我们选择C++作为第二开发语言的原因，是它拥有很多高质量、跨平台的图形库。我们最终选用Qt5作为图形化界面的开发库。但对于这门语言本身，我们没有任何期待。它糟糕透了。：）

内核与引擎，策略与机制的分离

降低软件复杂度的手段之一是实现策略与机制的分离。“策略”是指程序和用户之间交互的接口，或者说程序交互界面；“机制”则是程序内部的核心逻辑。将策略从机制中分离出来，将有极大的好处：

1. 两个部分得以分离测试。比起“机制”和“策略”粘合在一起的情况，将二者分离会允许开发者迅速定位策略的来源。这无疑缩小了调试纠错的检查范围。
2. “机制”和“策略”部分可以采用最适用的开发方式。这一点将在后文中看到。
3. 它提供了一种抽象屏障。这种实现方式让“策略”的实现者无需关注软件内核——即机制的内部细节，只需要明确和内部交互的接口或者协议。这极大地减少了开发者所需要记忆的事物，也减少了开发者的开发负担。

我们将展示我们是如何将内核与引擎，策略与机制干净地分开的。

Socket之刃

BSD开发的Unix套接字——socket，让两台计算机之间的互连变得异常简单。我们正是通过Socket，让内核与引擎干净地分离。
将两个部分分开的机制有很多，至少包括多进程、多线程和共享内存池、文件交互几种，但我们却最终选择了socket。对于这个选择，我们有充足的原因。
首先，我们立即抛却了文件交互方式。因为在本项目中，用户界面需要和程序内核频繁访问。通过文件机制显得过于脆弱。文件的正常打开和正常读写在频繁的交互中难以保证，并且文件交互不可保证临时文件在程序运行中的可靠性——文件太容易被外界因素所修改。因此使用文件非常不明智。
后来，我们进一步放弃了多线程与多进程编程。多线程是Bug密集之地，我们不得不通过锁机制来保证数据的唯一性，否则程序内部数据很容易被污染。在我的编程经验中，我曾经仿照Unix管道建立了一种内外线程互享机制，本质是一种消息传递。这种互享机制虽然比较稳健，但其复杂度大大超过了其他所有的实现。多进程和操作系统的联系太过密切，我们希望自己的程序有更高的可移植性和抽象程度，因此最终放弃。
我们最终选择了socket。使用这种机制，在内核和外壳之间建立一个临时连接，沿用TCP/IP协议进行数据传输。既干净，又高效。
此外，socket还赋予了我们将软件抽象成Server对Client服务的能力。事实上，我们程序的核心就是一个Server，而与用户的接口则是一个简单而干净的Client。

抽象的神力

为了提高socket服务的功能和性能，我们在transmitter模块中引入了MASS（Modularized Abstract Socket Server， 模块化抽象套接字服务器）。MASS是对Python所提供的Socket API的进一步抽象，每一个MASS都是对一个端口的连接绑定，可以在运行期全程提供连接服务。另外，其最大的要点是，每个MASS模块都携带一个只属于自己的内部自动机（Machine）。也就是说，每个传输服务器都可以具有自己独一无二的行为。所有服务器都是可以定制的，它可以对应不同的核心算法，实现不同的功能。
MASS服务器的引入，允许我们：

1. 同时提供多个服务器，允许多个客户端同时与服务端交互。
2. 运行时对服务器有完全的掌控，主动开闭服务器的行为得到技术支持。
3. 服务器具有高度的可定制性，不仅可以定制服务器的属性性质，还可以定制属于每个服务器单独的内部行为。
4. 多服务器同时运行，彼此相互正交，互不影响，提高了程序的稳定性和健壮性。

软件的许多功能的实现是与上面这些特性分不开的。服务器的可定制性看起来并不起眼，但它确是我们分别实现分配服务器（dispatcher）与普通服务器的技术基础。MASS允许我们将一个服务器本身抽象成一个对象，而决定这个对象行为的就是其所含的机器（Machine）。面向对象的抽象，与高度的模块化，成为transmitter模块得以稳健而强大的基础。

抽象→进化→繁殖

在transmitter模块开发初期，我的目的只是建立一个客户端与服务端相连相通的机制。因此当时的socket服务器还没有抽象成一个独立的对象，而是直接写在了程序运行的入口函数里。接着，意识到众多的操作（开启连接、连接绑定、关闭连接）都是直接围绕着socket服务器本身，因此抽象就发生了：socket server被抽象成了MASS，成为程序运行时临时生成的、一个包装好的对象。这个对象可以有独立的行为，这些行为与外界无关。进一步地，发现MASS的封装性良好，完全可以当成是运行时创建的一个简单的Python对象，那么，为什么我们不可以在运行时创造多个对象，从而建立多个服务器，允许多个客户端同时与服务端交互呢？这从理论上是绝对可行的。但实际开发过程中还是遇到了不少坎坷。
为了实现多个客户端同时连接，我们不得不开启多个端口（com）。起初，我默认了客户端知晓服务端开启了哪些端口，因此让客户端手动输入端口。但这显然并不是一个明智的选择——客户应该对服务器的底层实现细节无知，更别说知道服务端哪些端口是开着的。所以，我们必须引入一个机制，让服务端可以自动**分配**（dispatch）端口。把开着的、闲置的那些端口开放给服务端，从而允许客户端能够直接与服务端相连。

方法就是，在数个MASS之外，增设一个端口分配服务器（dispatcher server）。端口服务器会确保在所有普通服务器之前首先运行，它负责将能够使用的端口分配给客户端。
端口分配服务器本身的端口是确定的（在本项目中默认使用9999端口，可以更改。），客户端会选择连接这个端口，并且进入接受状态。在连接建立以后，端口分配服务器只会做一件事——寻找那些可以用的端口，直接把端口发给客户端，然后关闭自己。客户端在接受这个端口号之后关闭与分配器的连接，接着建立向所收到的端口的连接。而紧接着端口分配服务器会重启，重新将自己绑定在默认端口上。在整个程序工作的流程中，只有很短的时间里端口服务器是忙的，或者是断开的；因此可以认为端口分配器在程序运行的所有时间内都是处在等待绑定的空闲状态。
因此，我们只用了一个端口分配服务器，就完成了对客户端的端口分配工作。从理论上来说，一台服务器可以同时承载几万个端口的服务。

在端口分配服务器的问题解决之后，剩下的问题就是如何解决服务器的再利用与重启。对此我们引入了相应的法则和异常处理机制，让transmitter模块在程序运行的过程中始终轮询服务器池，一旦发现有端口损坏或者被断开，就自动重置该端口为空闲状态，并且重新将其开启。

MASS的引入，让我们的软件具有同时负载多个客户端的能力。这更接近现实中的地图导航系统。事实上，我们为每个端口服务器都分配了专有的线程，加上transmitter本身管理机制的线程与核心模块的线程，在程序运行过程中，十个以上线程同时开启的情况经常会出现。但是MASS的引入让一切服务器都彼此正交而无干扰，因此即使是并行操作，我们也不需要任何锁机制。在一个服务器因为故障而崩溃的时候，其他服务器仍然可以照常运行，而transmitter管理机制能够重启这个坏掉的服务器，从而重新回收垃圾资源，并让程序风雨无阻的运行。这就是抽象封装带给我们的神力所在。

模块化——我们的接口不发黏

模块化设计是降低程序内部复杂度的一个重要手段。一个良好的设计应该是让每个模块彼此正交，毫不粘连。在我们的项目中，模块也被划分成了足够细而不会太细的粒度，让每个模块各司其职，互不干扰。但是“模块”并不是几个独立的程序，不同的模块之间彼此也需要广泛的交流。模块之间定义良好的接口让模块交流变得高效而简单。但是，在一些特定的场合，即使是使用定义明确的接口，我们也不能保证模块之间的干扰降到最低。接口本身也需要占用代码，而这些来自外部的代码将是程序中最不结实的积木。因为，负责不同模块开发的往往是不同的人，他们彼此对对方的工作状况并不知晓。即使接口使用了最简洁的语言，开发在定义接口的时候也是使用了自己所理解的意义，往往我们对自己的工作了如指掌且充满自信，而换到别人的眼里，却千疮百孔、疑点重重。

因此，总结来说，造成接口处bug云集的最大原因是，一个模块对模块外部的世界是不可知的。模块化设计要求我们将模块划分到正交的关系，因此要求我们将一个模块独立化，对自己十分清楚，对外界就像是盲人。同时，开发者对于自己所管辖的那个域以外的部分也一无所知。这“双盲”局面造成的结果就是接口调用处的代码可能动作不可预知，行为难以捉摸。

解决这个问题的方法是：

1. 把当前所有对外界模块的调用都变得透明、运行时可知。
   要了解程序运行时当前模块调用了哪些外界函数，开发者常常需要调用debug程序，以单步调试的方式逐行分析。这从速度上说并非明智。要想知道模块调用了哪些内容，开发者不得不把程序暂停下来，或者重新运行一遍。而倘若所有的模块调用都可以以某种运行日志的方式输出出来，加上相应的时间戳，程序员就很容易在离线（不面对代码）的时候，通过浏览日志来找出是哪些地方出现了问题。
2. 不显式地调用外界的代码，而是实现内部的协议转换。
   既然当前模块对外界模块是不可知的，那么它就应该对外界彻底不可见（blind），而不应该使用接口约定的方式来捕获外界的光。要达到这一点，我们必须建立一个机制，**让外面的模块知道当前模块想要调用什么，而当前模块并不知道自己所要调用的语句在其他模块中是以什么方式存在。**很显然，我们需要一个协议，更具体来说，是一个解释器。当前模块的代码以自己的语言把代码送进解释器，然后解释器负责把代码翻译成外界模块的语言。这虽然也需要内外模块的开发者合作，共商协议，但它允许我们集中处理这些问题，而不是将接口设计弥散在整个程序里。
3. 尽量减少两个模块之间的接触面积。一旦切口变小了，所有bug涌现的方式就会十分有限。

为了实现这个思路，引入mailer机制。我们约定：

- 每个模块有一个属于自己的“信箱”（mailbox），负责接收来自外界的函数调用请求。
- 信箱可以收发“邮件”，“邮件”的内容是对当前模块中某个函数的调用。在每个模块收到一个“邮件”的时候，这个“邮件”里的函数并非立即执行。模块有权利检查这个“邮件”的内容，并且由它来决定是否要执行里面的内容。这实现了调用权与执行权的分离。外界模块有能力请求该模块执行该模块所管辖的代码，而真正有权利决定代码要不要执行的，正是代码的管辖者，当前模块本身。
- 邮件中的函数由一个暴露在外的解释器来书写。也就是说，其他模块的函数对当前模块不可知，但是一个解释器可以将当前模块想要执行的命令翻译成正确的函数调用。

我们首先将这个机制应用在核心模块与发送器（transmitter）模块的交互上。由于transmitter只是作为一个收发器而存在，因此所有客户端想要执行的命令都必须被翻译成对核心模块的函数的调用。因此，两个模块的交互动作将会非常多。mailer机制很好地将这些调用都集中到一起，减少了两个模块之间的接触面积。此外，我们得益于这个机制的引入，我们可以将一切运行时发生的信息都记录下来。无论是所接受到的客户端发来的信息，还是transmitter向core的函数调用，都被一网打尽。程序整个运作流程都变得透明可见，它可以详尽的记录下所有有用的信息。

总结

以上即是我对当前开发进度下程序基础架构和机制设计的简略概述。我们引入的一切机制无疑在降低程序复杂度、提升程序性能方面立下了汗马功劳。正因抽象服务器的存在，我们得以建立多端口S-C服务框架，并稳健地同时管理多个服务器。也正因mailer机制的引入，两个关键模块之间的接口变得简单，我们甚至因此而获得了记录更详细的运行日志的能力。
但是目前所采用的机制尚有些不足。一个很大的缺点就是mailer机制本身具有较高的复杂度，让我们难以将这个机制推广。但我相信我们可以通过建立更高层的抽象，将这个机制变得更加易用，更加高效。

2016/4/14
Map-Walker小组