3.1 设计模式介绍

3.1.1 简单工厂模式

是什么：在简单工厂模式中，可以根据传递的参数不同，返回不同类的实例。
解决哪些问题：解决了对象的创建问题。
什么时候使用：用于已知某些条件后，对类的选择，而这些类都是同一父类的子类。 简单工厂模式最大的优点在于实现对象的创建和对象的使用分离，但是如果产品过多时，会导致工厂代码非常复杂。简单工厂模式的要点就在于当你需要什么，只需要传入一个正确的参数，就可以获取你所需要的对象，而无须知道其创建细节。

3.1.2 Java实现

重点：步骤三---》创建工厂类
我们将创建一个 *Shape* 接口和实现 *Shape* 接口的实体类。下一步是定义工厂类 *ShapeFactory*。
使用 *ShapeFactory* 来获取 *Shape* 对象。它将向 *ShapeFactory* 传递信息（*CIRCLE / RECTANGLE / SQUARE*），以便获取它所需对象的类型。

```
public abstract class Operation {
    public double numberA;
    public double numberB;

    public abstract double result();
}

public class OperationAdd extends Operation {

    @Override
    public double result() {
        return numberA + numberB;
    }

}

public class OperationSub extends Operation {

    @Override
    public double result() {
        return numberA - numberB;
    }

}


public class OperationMul extends Operation {
```

```
    @Override
    public double result() {
        return numberA * numberB;
    }

}

public class OperationDiv extends Operation {

    @Override
    public double result() {
        if (numberB == 0) {
            throw new RuntimeException("divided by 0");
        }
        return numberA / numberB;
    }

}

public class OperationFactory {
    public static Operation createOperation(char operator) {
        Operation operation = null;

        switch (operator) {
        case '+':
            operation = new OperationAdd();
            break;
        case '-':
            operation = new OperationSub();
            break;
        case '*':
            operation = new OperationMul();
            break;
        case '/':
            operation = new OperationDiv();
            break;
        default:
            throw new RuntimeException("unsupported operation");
        }

        return operation;
    }
}

public class Calculator {
    public static void main(String[] args) {
        Operation operation;
        char operator;
```

```
        operator = '+';
        operation = OperationFactory.createOperation(operator);
        operation.numberA = 1.2;
        operation.numberB = 2.3;

        System.out.println(operation.result());
    }
}
```

3.1.3 Python实现

```
from abc import abstractmethod

class Operationx(object):
    def __init__(self):
        self.__numberA=0
        self.__numberB=0

    def getNumberA(self):
        return self.__numberA

    def getNumberB(self):
        return self.__numberB

    def setNumberA(self,value):
        self.__numberA=value

    def setNumberB(self,value):
        self.__numberB=value

    numberA = property(getNumberA, setNumberA)
    numberB = property(getNumberB, setNumberB)

    @abstractmethod
    def get_result(self):
        result=0
        return result

class OperatinAdd(Operationx):
    def get_result(self):
        result=self.getNumberA()+self.getNumberB()
        return str(result)

class OperationSub(Operationx):
    def get_result(self):
        result=self.getNumberA()-self.getNumberB()
```

```
        return str(result)

class OperationMul(Operationx):
    def get_result(self):
        return str(self.getNumberA()*self.getNumberB())

class OperationDiv(Operationx):
    def get_result(self):
        try:
            return str(self.getNumberA()/self.getNumberB())
        except ZeroDivisionError as erro:
            return "不能除以0! "

class SimpleOperationFactory:
    @staticmethod
    def createOperate(operate):
        if operate == "+":
            operator = OperatinAdd()
        elif operate == "-":
            operator = OperationSub()
        elif operate == "*":
            operator = OperationMul()
        elif operate == "/":
            operator = OperationDiv()
        else:
            operator="输入的算符不正确"
        return operator

if __name__=="__main__":
    op = input("请输入操作符+、-、*、/: ")
    operator=SimpleOperationFactory.createOperate(op)
    if not (isinstance(operator,str)):
        opa = eval(input("请输入数字A: "))
        opb = eval(input("请输入数字B: "))
        operator.numberA=opa
        operator.numberB=opb
        result=operator.get_result()
    else:
        result=operator
    print("计算结果: "+result)
```

在ROS的世界里，最小的进程单元就是节点（node）。一个软件包里可以有多个可执行文件，可执行文件在运行之后就成了一个进程(process)，这个进程在ROS中就叫做**节点**。从程序角度来说，node就是一个可执行文件（通常为C++编译生成的可执行文件、Python脚本）被执行，加载到了内存之中；从功能角度来说，通常一个node负责者机器人的某一个单独的功能。由于机器人的功能模块非常复杂，我们往往不会把所有功能都集中到一个node上，而会采用分布式的方式，把鸡蛋放到不同的篮子里。例如有一个node来控制底盘轮子的运动，有一个node驱动摄像头获取图像，有一个node驱动激光雷达，有一个node根据传感器信息进行路径规划……这样做可以降低程序发生崩溃的可能性，试想一下如果把所有功能都写到一个程序中，模块间的通信、异常处理将会很麻烦。

我们在1.4节打开了小海龟的运动程序和键盘控制程序，在1.5节同样启动了键盘运动程序，这每一个程序便是一个node。ROS系统中不同功能模块之间的通信，也就是节点间的通信。我们可以把键盘控制替换为其他控制方式，而小海龟运动程序、机器人仿真程序则不用变化。这样就是一种模块化分工的思想。

3.1.2 Master

由于机器人的元器件很多，功能庞大，因此实际运行时往往会运行众多的node，负责感知世界、控制运动、决策和计算等功能。那么如何合理的进行调配、管理这些node？这就要利用ROS提供给我们的节点管理器master, master在整个网络通信架构里相当于管理中心，管理着各个node。node首先在master处进行注册，之后master会将该node纳入整个ROS程序中。node之间的通信也是先由master进行“牵线”，才能两两的进行点对点通信。当ROS程序启动时，第一步首先启动master，由节点管理器处理依次启动node。

3.1.3 启动master和node

当我们要启动ROS时，首先输入命令:

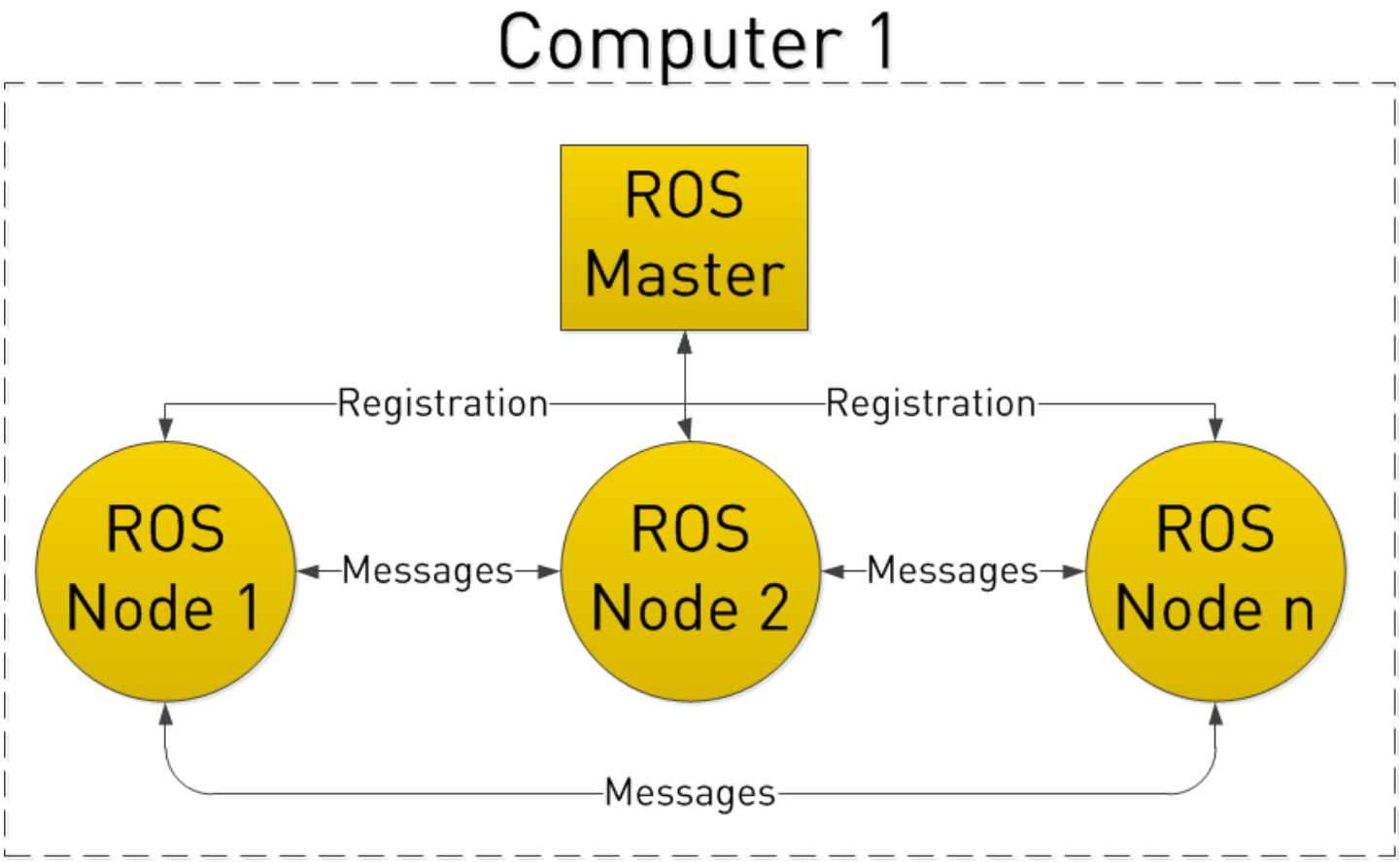
```
$ roscore
```

此时ROS master启动，同时启动的还有`rosout`和`parameter server`,其中`rosout`是负责日志输出的一个节点，其作用是告知用户当前系统的状态，包括输出系统的error、warning等等，并且将log记录于日志文件中，`parameter server`即是参数服务器，它并不是一个node，而是存储参数配置的一个服务器，后文我们会单独介绍。每一次我们运行ROS的节点前，都需要把master启动起来，这样才能够让节点启动和注册。

master之后，节点管理器就开始按照系统的安排协调进行启动具体的节点。节点就是一个进程，只不过在ROS中它被赋予了专用的名字里——node。在第二章我们介绍了ROS的文件系统，我们知道一个package中存放着可执行文件，可执行文件是静态的，当系统执行这些可执行文件，将这些文件加载到内存中，它就成为了动态的node。具体启动node的语句是:

```
$ rosrun pkg_name node_name
```

通常我们运行ROS，就是按照这样的顺序启动，有时候节点太多，我们会选择用launch文件来启动，下一小节会有介绍。 Master、Node之间以及Node之间的关系如下图所示:



3.1.3 rosrun和rosnode命令

rosrun命令的详细用法如下:

```
$ rosrun [--prefix cmd] [--debug] pkg_name node_name [ARGS]
```

rosrun将会寻找PACKAGE下的名为EXECUTABLE的可执行程序，将可选参数ARGS传入。 例如在GDB下运行ros程序:

```
$ rosrun --prefix 'gdb -ex run --args' pkg_name node_name
```

rosnode命令的详细作用列表如下:

rosnode命令	作用
`rosnode list`	列出当前运行的node信息
`rosnode info node_name`	显示出node的详细信息
`rosnode kill node_name`	结束某个node
`rosnode ping`	测试连接节点

rosnode命令	作用
`rosnode machine`	列出在特定机器或列表机器上运行的节点
`rosnode cleanup`	清除不可到达节点的注册信息

以上命令中常用的为前三个，在开发调试时经常会需要查看当前node以及node信息，所以请记住这些常用命令。如果你想不起来，也可以通过`rosnode help`来查看`rosnode`命令的用法。