实战Java高并发程序设计第7周

DATAGURU专业数据分析社区

法律声明

【声明】本视频和幻灯片为炼数成金网络课程的教学资料，所有资料只能在课程内使用，不得在课程以外范围散播，违者将可能被追究法律和经济责任。

课程详情访问炼数成金培训网站

http://edu.dataguru.cn

炼数成金逆向收费式网络课程

- Dataguru（炼数成金）是专业数据分析网站，提供教育，媒体，内容，社区，出版，数据分析业务等服务。我们的课程采用新兴的互联网教育形式，独创地发展了逆向收费式网络培训课程模式。既继承传统教育重学习氛围，重竞争压力的特点，同时又发挥互联网的威力打破时空限制，把天南地北志同道合的朋友组织在一起交流学习，使到原先孤立的学习个体组合成有组织的探索力量。并且把原先动辄成千上万的学习成本，直线下降至百元范围，造福大众。我们的目标是：低成本传播高价值知识，构架中国第一的网上知识流转阵地。
- 关于逆向收费式网络的详情，请看我们的培训网站 http://edu.dataguru.cn

内容提要

- 什么是设计模式
- 单例模式
- 不变模式
- Future模式
- 生产者消费者

什么是设计模式

- 在软件工程中，设计模式（design pattern）是对软件设计中普遍存在（反复出现）的各种问题，所提出的解决方案。这个术语是由埃里希·伽玛（Erich Gamma）等人在1990年代从建筑设计领域引入到计算机科学的。
- Richard Helm, Ralph Johnson ,John Vlissides （Gof）
- 《设计模式：可复用面向对象软件的基础》 收录 23种模式
  - 观察者模式
  - 策略模式
  - 装饰者模式
  - 享元模式
  - 模板方法
  - .....

什么是设计模式

- 架构模式
  - MVC
  - 分层
- 设计模式
  - 提炼系统中的组件
- 代码模式（成例 Idiom）
  - 低层次，与编码直接相关
  - 如DCL

```
class Person {
  String name;
  int birthYear;
  byte[] raw;

  public boolean equals(Object obj) {
    if (!obj instanceof Person)
      return false;

    Person other = (Person)obj;
    return name.equals(other.name)
        && birthYear == other.birthYear
        && Arrays.equals(raw, other.raw);
  }

  public int hashCode() { ... }
}
```

单例模式

- 单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为
- 比如：全局信息配置

单例模式

```
1 public class Singleton {
2         private Singleton(){
3                 System.out.println("Singleton is create");
4         }
5         private static Singleton instance = new Singleton();
6         public static Singleton getInstance() {
7                 return instance;
8         }
9 }
```

何时产生实例 不好控制

单例模式

```
public class Singleton {
        public static int STATUS=1;
        private Singleton(){
                System.out.println("Singleton is create");
        }
        private static Singleton instance = new Singleton();
        public static Singleton getInstance() {
                return instance;
        }
}
```

```
System.out.println(Singleton.STATUS);
```

```
Singleton is create
1
```

单例模式

```
01 public class LazySingleton {
02         private LazySingleton() {
03                 System.out.println("LazySingleton is create");
04         }
05         private static LazySingleton instance = null;
06         public static synchronized LazySingleton getInstance() {
07                 if (instance == null)
08                         instance = new LazySingleton();
09                 return instance;
10         }
11 }
```

单例模式

```
01 public class StaticSingleton {
02         private StaticSingleton(){
03                 System.out.println("StaticSingleton is create");
04         }
05         private static class SingletonHolder {
06                 private static StaticSingleton instance = new StaticSingleton();
07         }
08         public static StaticSingleton getInstance() {
09                 return SingletonHolder.instance;
10         }
11 }
```

不变模式

- 一个类的内部状态创建后，在整个生命期间都不会发生变化时，就是不变类
- 不变模式不需要同步

不变模式

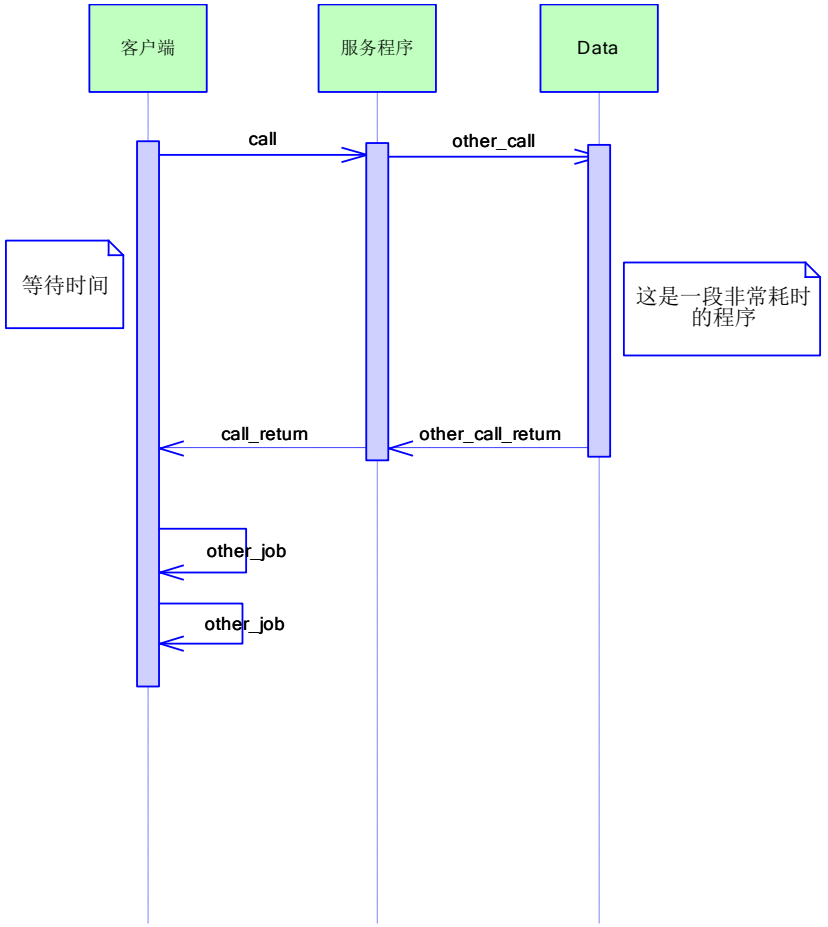
```
public final class Product {
        //确保无子类
        private final String no;
        //私有属性，不会被其他对象获取
        private final String name;
        //final保证属性不会被2次赋值
        private final double price;

        public Product(String no, String name, double price) {        //在创建对象时，必须指定数据
                super();
                        //因为创建之后，无法进行修改
                this.no = no;
                this.name = name;
                this.price = price;
        }

        public String getNo() {
                return no;
        }
        public String getName() {
                return name;
        }
        public double getPrice() {
                return price;
        }
}
```

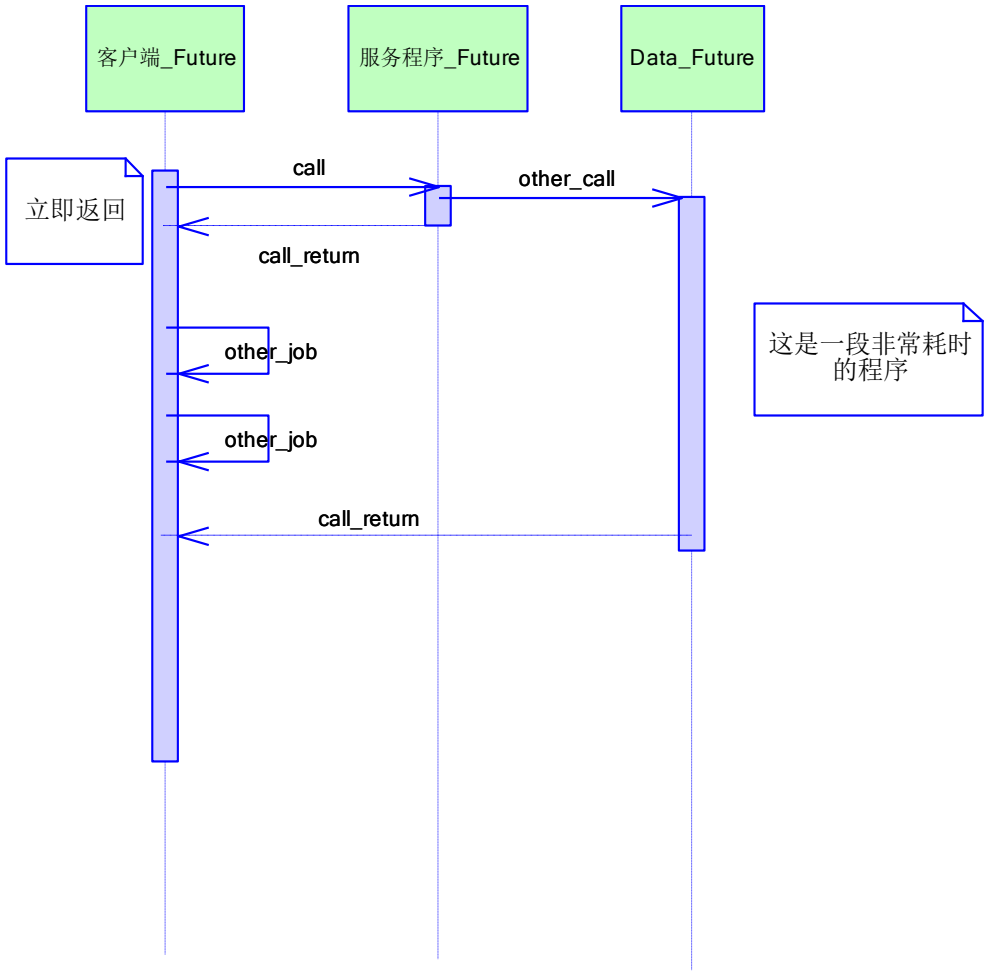
不变模式

- java.lang.String
- java.lang.Boolean
- java.lang.Byte
- java.lang.Character
- java.lang.Double
- java.lang.Float
- java.lang.Integer
- java.lang.Long
- java.lang.Short

Future模式

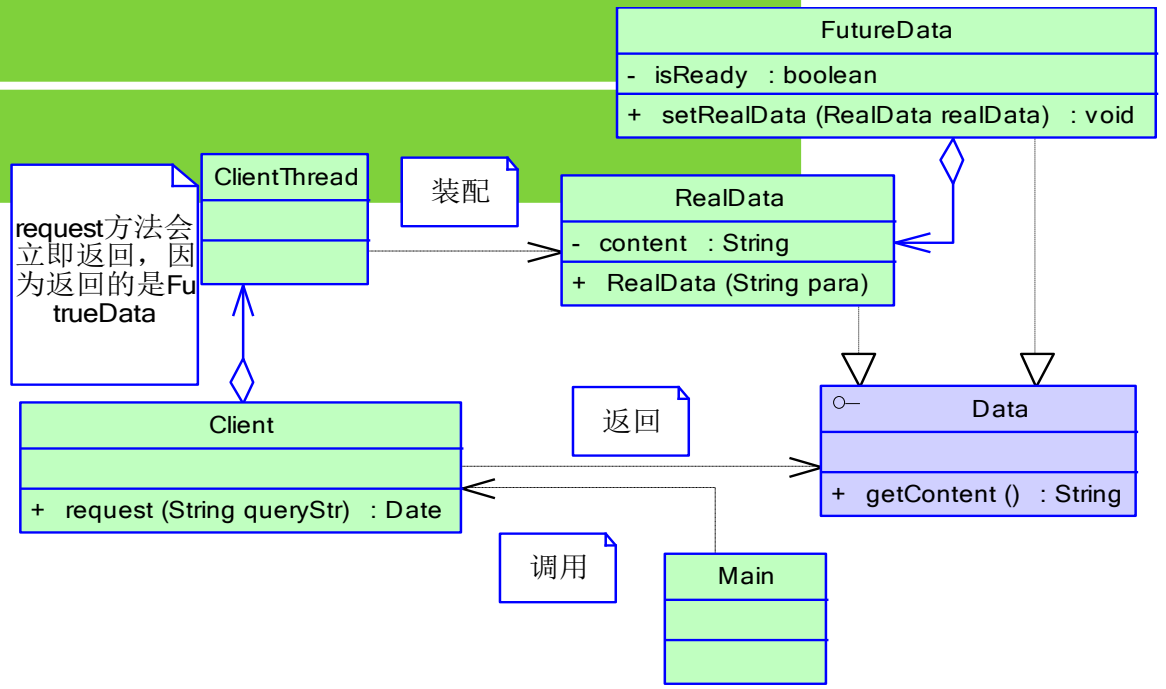
- 核心思想是异步调用





Future模式

参与者	作用
Main	系统启动，调用Client发出请求
Client	返回Data对象，立即返回FutureData，并开启ClientThread线程装配RealData
Data	返回数据的接口
FutureData	Future数据，构造很快，但是是一个虚拟的数据，需要装配RealData
RealData	真实数据，其构造是比较慢的



Future模式

```
public interface Data {
    public String getResult ();
}
```

```
public class FutureData implements Data {
    protected RealData realdata = null;                    //FutureData是RealData的包装
    protected boolean isReady = false;
    public synchronized void setRealData(RealData realdata) {
        if (isReady) {
            return;
        }
        this.realdata = realdata;
        isReady = true;
        notifyAll();                                       //RealData已经被注入，通知getResult()
    }
    public synchronized String getResult() {               //会等待RealData构造完成
        while (!isReady) {
            try {
                wait();                                    //一直等待，知道RealData被注入
            } catch (InterruptedException e) {
            }
        }
        return realdata.result;                                        //由RealData实现
    }
}
```

Future模式

```
public class RealData implements Data {
    protected final String result;
    public RealData(String para) {
            //RealData的构造可能很慢，需要用户等待很久，这里使用sleep模拟
            StringBuffer sb=new StringBuffer();
        for (int i = 0; i < 10; i++) {
            sb.append(para);
          try {
            //这里使用sleep，代替一个很慢的操作过程
             Thread.sleep(100);
          } catch (InterruptedException e) {
          }
        }
        result =sb.toString();
    }
    public String getResult() {
        return result;
    }
}
```

Future模式

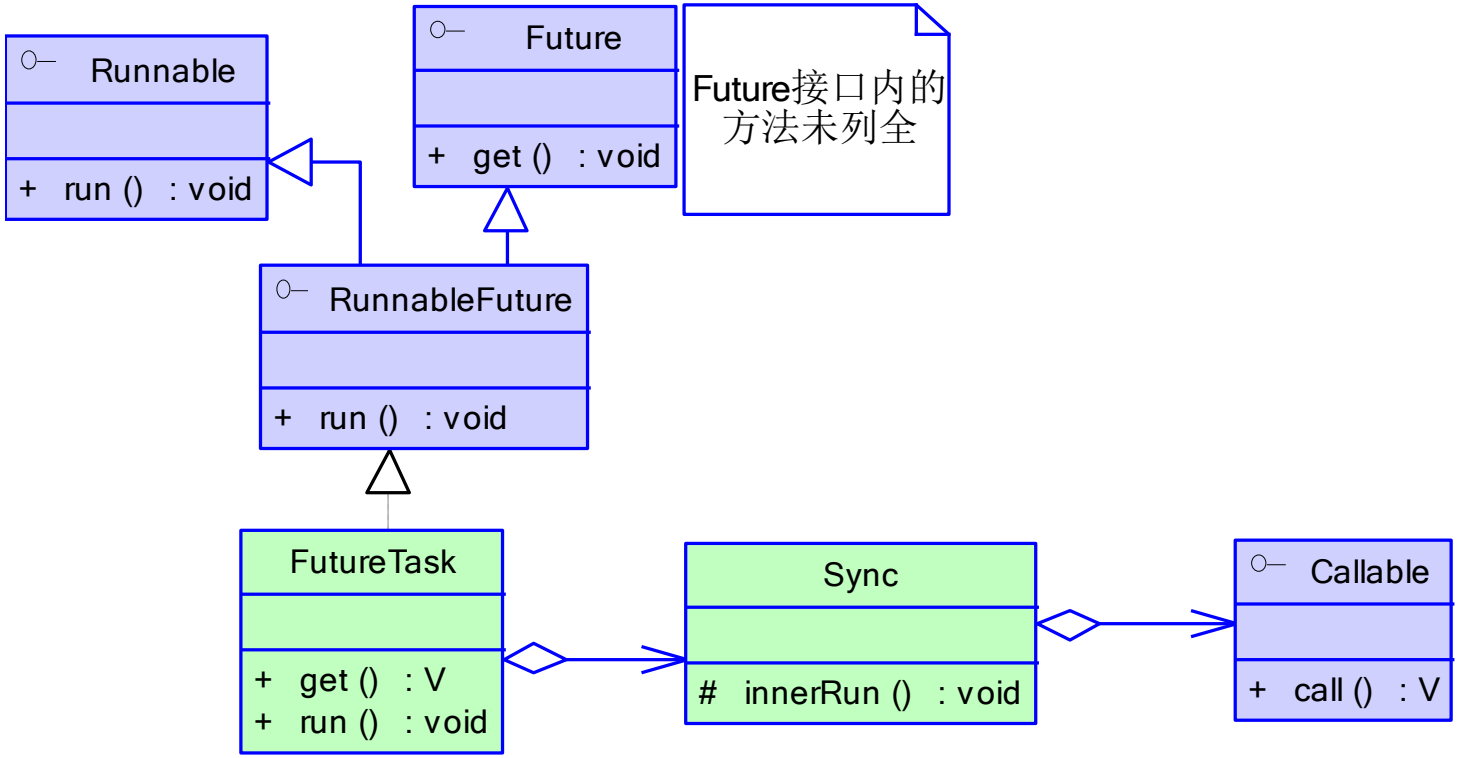
```
public class Client {
    public Data request(final String queryStr) {
        final FutureData future = new FutureData();
        new Thread() {
            public void run() {// RealData的构建很慢，

             //所以在单独的线程中进行
                RealData realdata = new RealData(queryStr);
                future.setRealData(realdata);
            }
        }.start();
        return future; // FutureData会被立即返回
    }
}
```

Future模式

```
public static void main(String[] args) {
    Client client = new Client();
    //这里会立即返回，因为得到的是FutureData而不是RealData
    Data data = client.request("name");
    System.out.println("请求完毕");
    try {
    //这里可以用一个sleep代替了对其他业务逻辑的处理
    //在处理这些业务逻辑的过程中，RealData被创建，从而充分利用了等待时间
        Thread.sleep(2000);
    } catch (InterruptedException e) {
    }
            //使用真实的数据
    System.out.println("数据 = " + data.getResult());
}
```

Future模式

- JDK对Future模式的支持



Future模式

```
01 public class RealData implements Callable<String> {
02     private String para;
03     public RealData(String para){
04         this.para=para;
05     }
06         @Override
07         public String call() throws Exception {
08
09         StringBuffer sb=new StringBuffer();
10        for (int i = 0; i < 10; i++) {
11         sb.append(para);
12           try {
13               Thread.sleep(100);
14           } catch (InterruptedException e) {
15           }
16        }
17        return sb.toString();
18          }
19 }
```

Future模式

```
01 public class FutureMain {
02     public static void main(String[] args) throws InterruptedException, ExecutionException {
03             //构造FutureTask
04         FutureTask<String> future = new FutureTask<String>(new RealData("a"));
05         ExecutorService executor = Executors.newFixedThreadPool(1);
06         //执行FutureTask，相当于上例中的 client.request("a") 发送请求
07         //在这里开启线程进行RealData的call()执行
08         executor.submit(future);
09
10         System.out.println("请求完毕");
11         try {
12         //这里依然可以做额外的数据操作，这里使用sleep代替其他业务逻辑的处理
13             Thread.sleep(2000);
14         } catch (InterruptedException e) {
15         }
16         //相当于data.getResult ()，取得call()方法的返回值
17         //如果此时call()方法没有执行完成，则依然会等待
18         System.out.println("数据 = " + future.get());
19     }
20 }
```

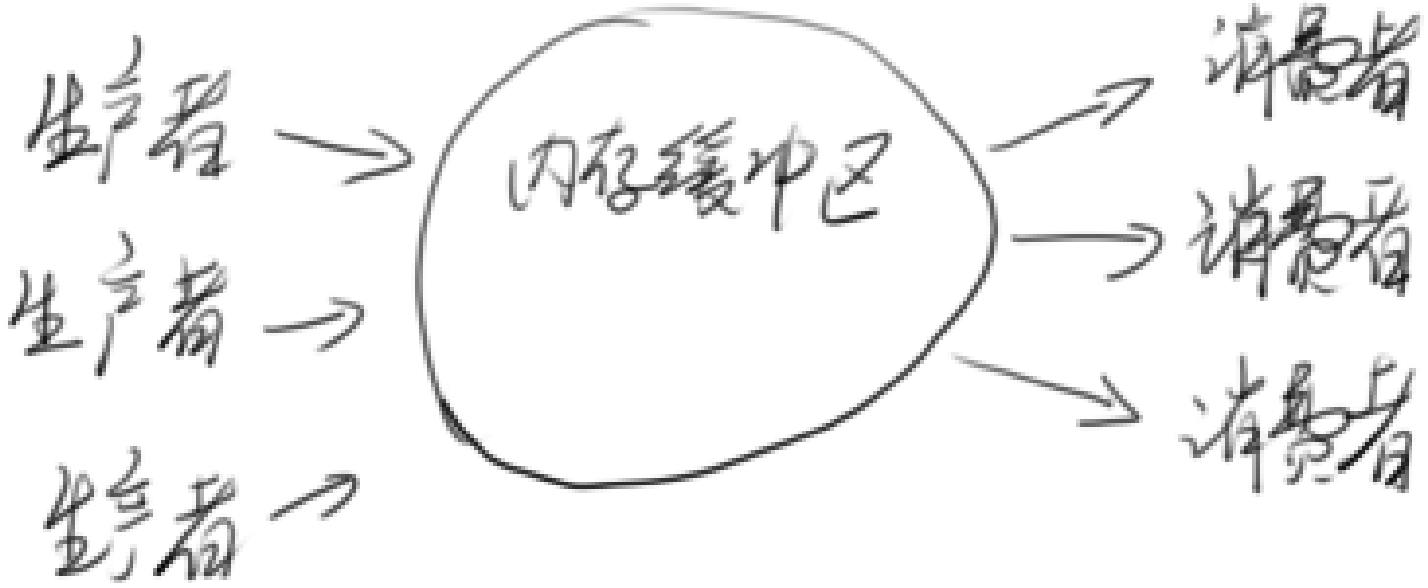
Future模式

```
public class FutureMain2 {
    public static void main(String[] args) throws InterruptedException, ExecutionException {

        ExecutorService executor = Executors.newFixedThreadPool(1);
        //执行FutureTask，相当于上例中的 client.request("a") 发送请求
        //在这里开启线程进行RealData的call()执行
        Future<String> future=executor.submit(new RealData("a"));

        System.out.println("请求完毕");
        try {
        //这里依然可以做额外的数据操作，这里使用sleep代替其他业务逻辑的处理
            Thread.sleep(2000);
        } catch (InterruptedException e) {
        }
        //相当于data.getResult ()，取得call()方法的返回值
        //如果此时call()方法没有执行完成，则依然会等待
        System.out.println("数据 = " + future.get());
    }
}
```

生产者消费者模式

- 生产者-消费者模式是一个经典的多线程设计模式。它为多线程间的协作提供了良好的解决方案。在生产者-消费者模式中，通常由两类线程，即若干个生产者线程和若干个消费者线程。生产者线程负责提交用户请求，消费者线程则负责具体处理生产者提交的任务。生产者和消费者之间则通过共享内存缓冲区进行通信。

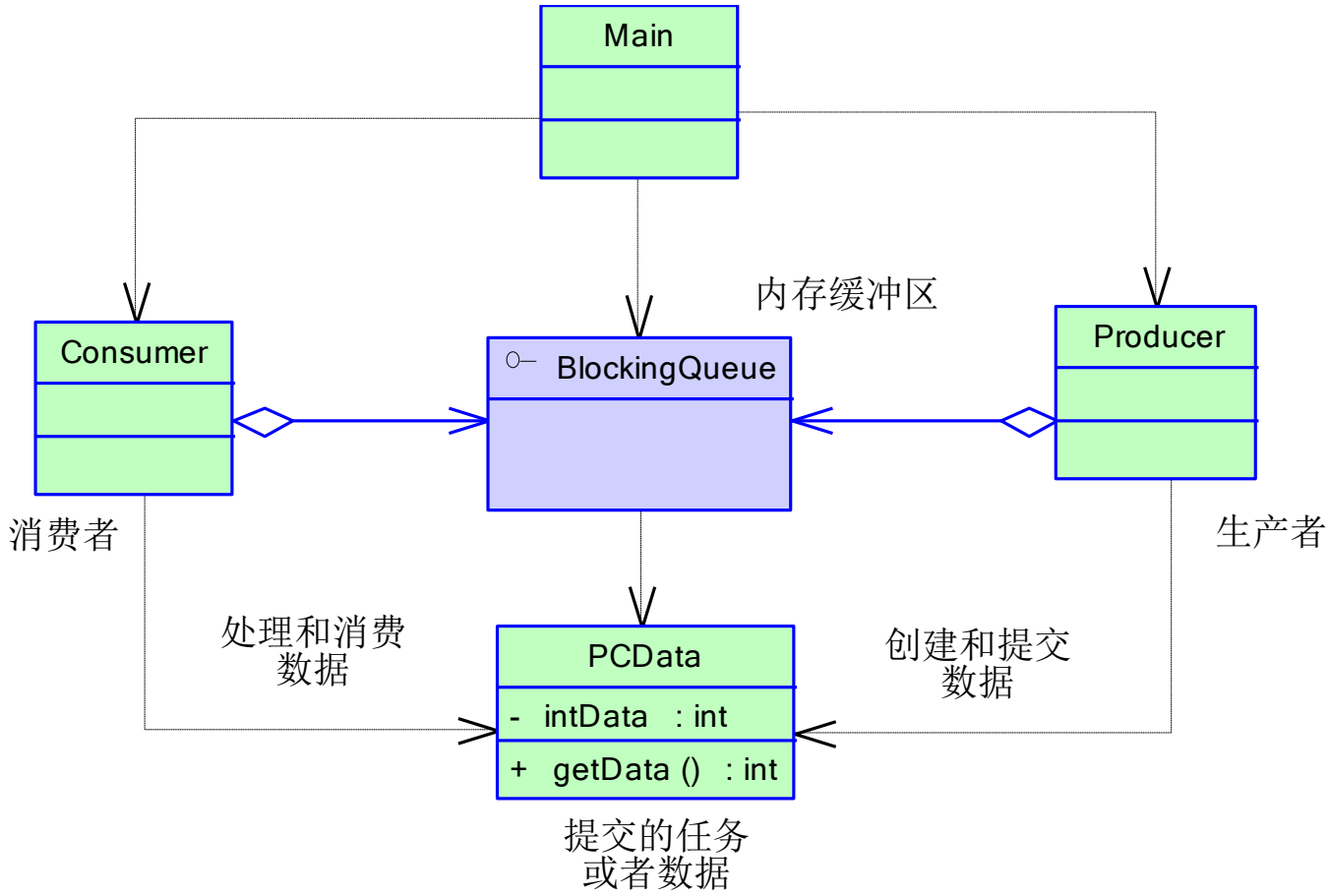
生产者消费者模式



生产者消费者模式

角色	作用
生产者	用于提交用户请求，提取用户任务，并装入内存缓冲区
消费者	在内存缓冲区中提取并处理任务
内存缓冲区	缓存生产者提交的任务或数据，供消费者使用
任务	生成者向内存缓冲区提交的数据结构。
Main	使用生产者和消费者的客户端

生产者消费者模式


Main
Consumer
BlockingQueue
Producer
内存缓冲区
消费者
生产者
处理和消费
数据
创建和提交
数据
PCData
- intData : int
+ getData () : int
提交的任务
或者数据

生产者消费者模式

```
while (isRunning) {
            Thread.sleep(r.nextInt(SLEEPTIME));
            data = new PCData(count.incrementAndGet());
            //构造任务数据
            System.out.println(data+" is put into queue");
            if (!queue.offer(data, 2, TimeUnit.SECONDS)) {
            //提交数据到缓冲区中
                        System.err.println("failed to put data : " + data);
            }
}
```

```
private BlockingQueue<PCData> queue;
```

```
while(true){
            PCData data = queue.take();
            //提取任务
            if (null != data) {
            int re = data.getData() * data.getData();      //计算平方

            System.out.println(MessageFormat.format("{0}*{1}={2}",
                                                data.getData(),
data.getData(), re));
                        Thread.sleep(r.nextInt(SLEEPTIME));
            }
}
```

Thanks

FAQ时间