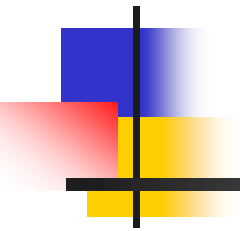
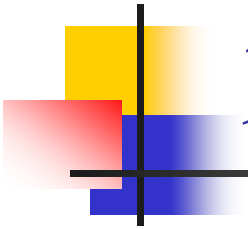


第4章 分布式对象和远程调用



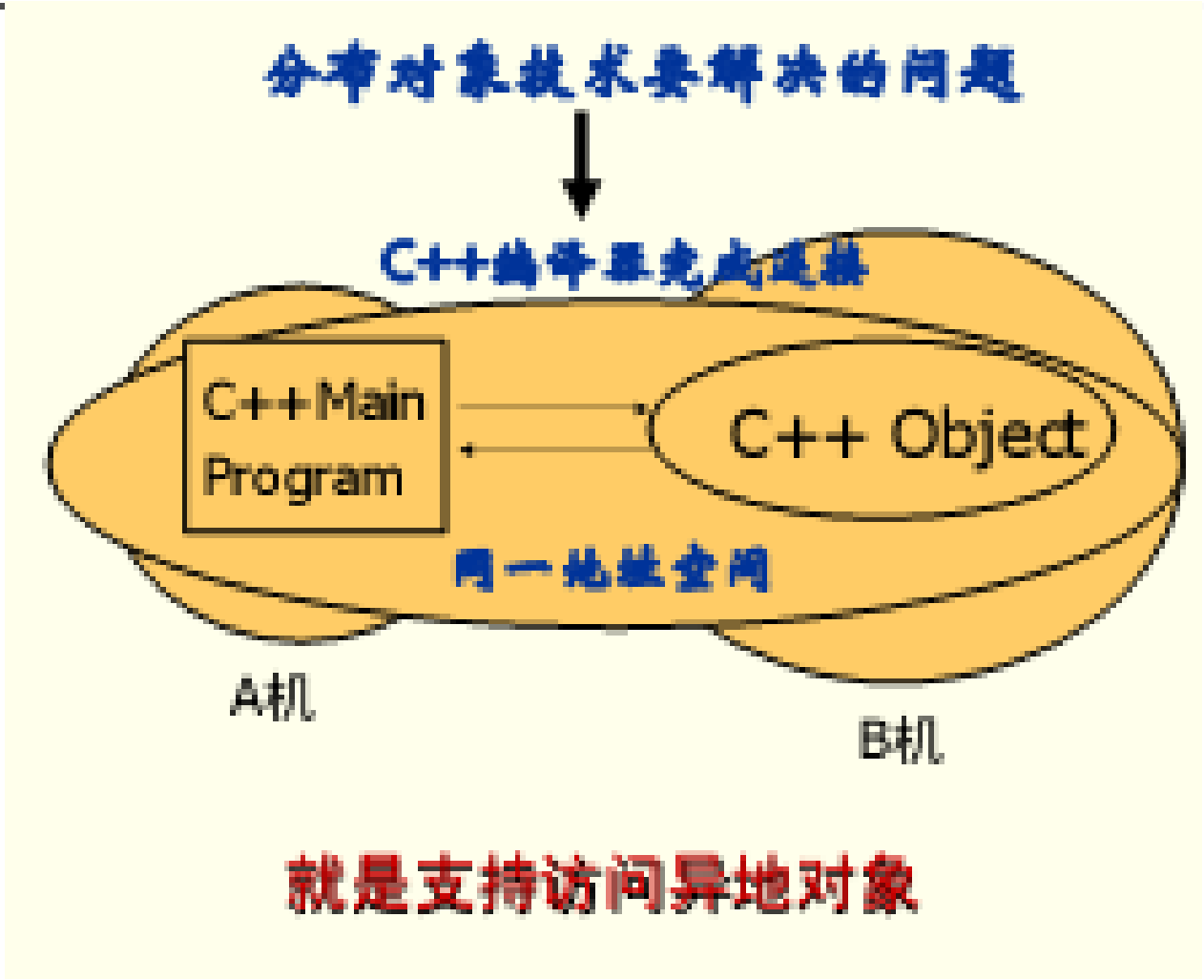
第4章:分布式对象和远程调用

- 引言
- 分布式对象间的通信
- 远程过程调用Remote procedure call
- 事件与通知Events and notifications
- Java RMI 实例研究
- 总结

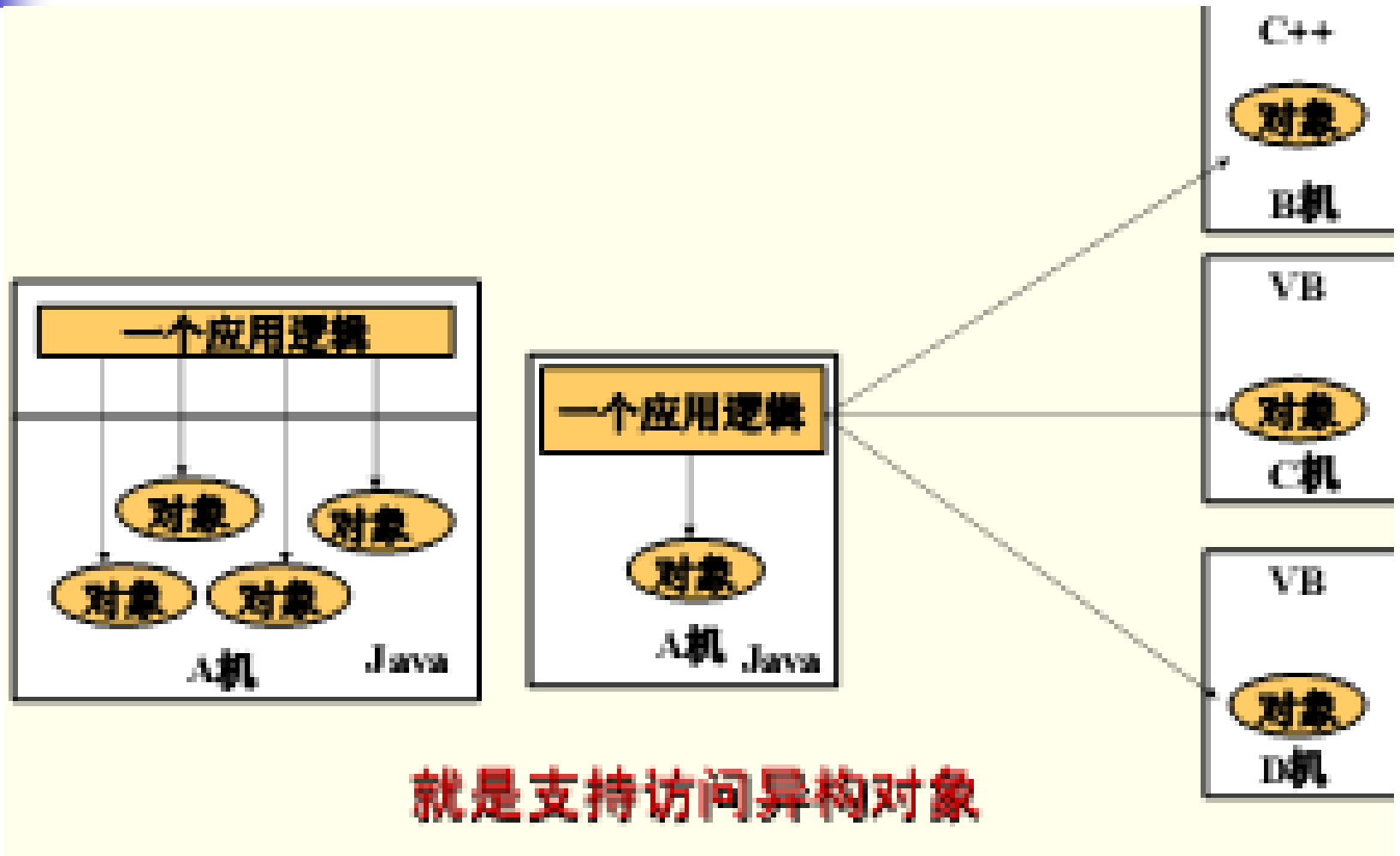
分布对象技术
要解决的基本问题

- 分布式系统的客户/服务器模型
- N层客户/服务器模型

分布对象技术
要解决的基本问题

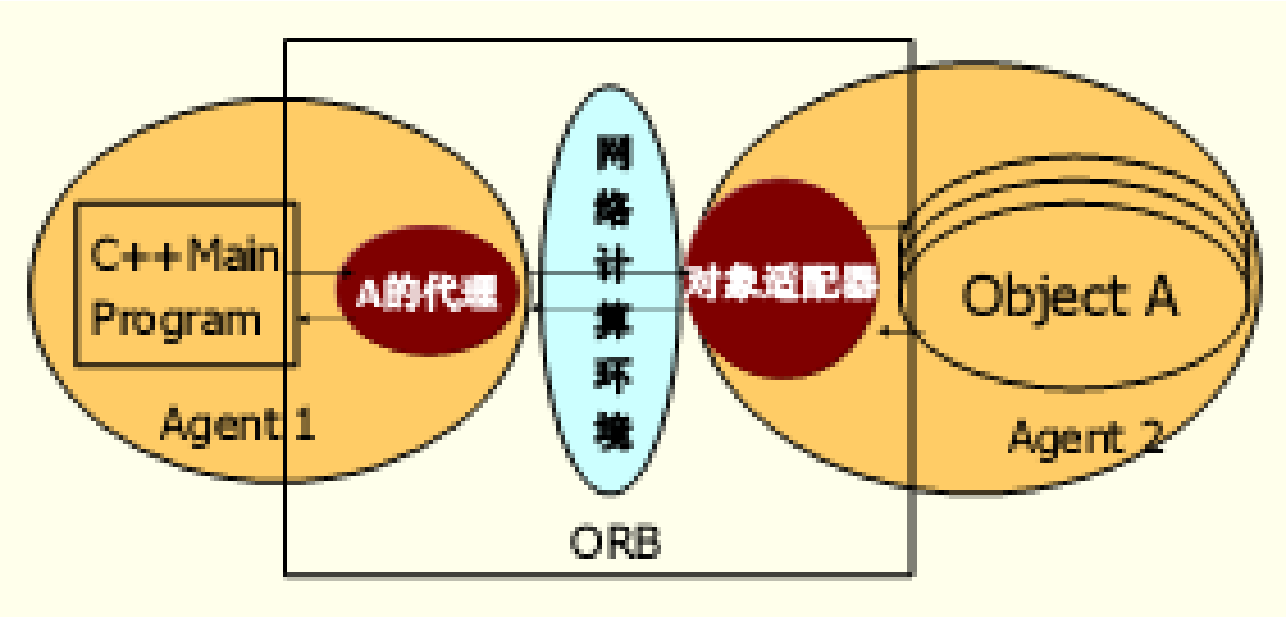


分布对象技术
要解决的基本问题

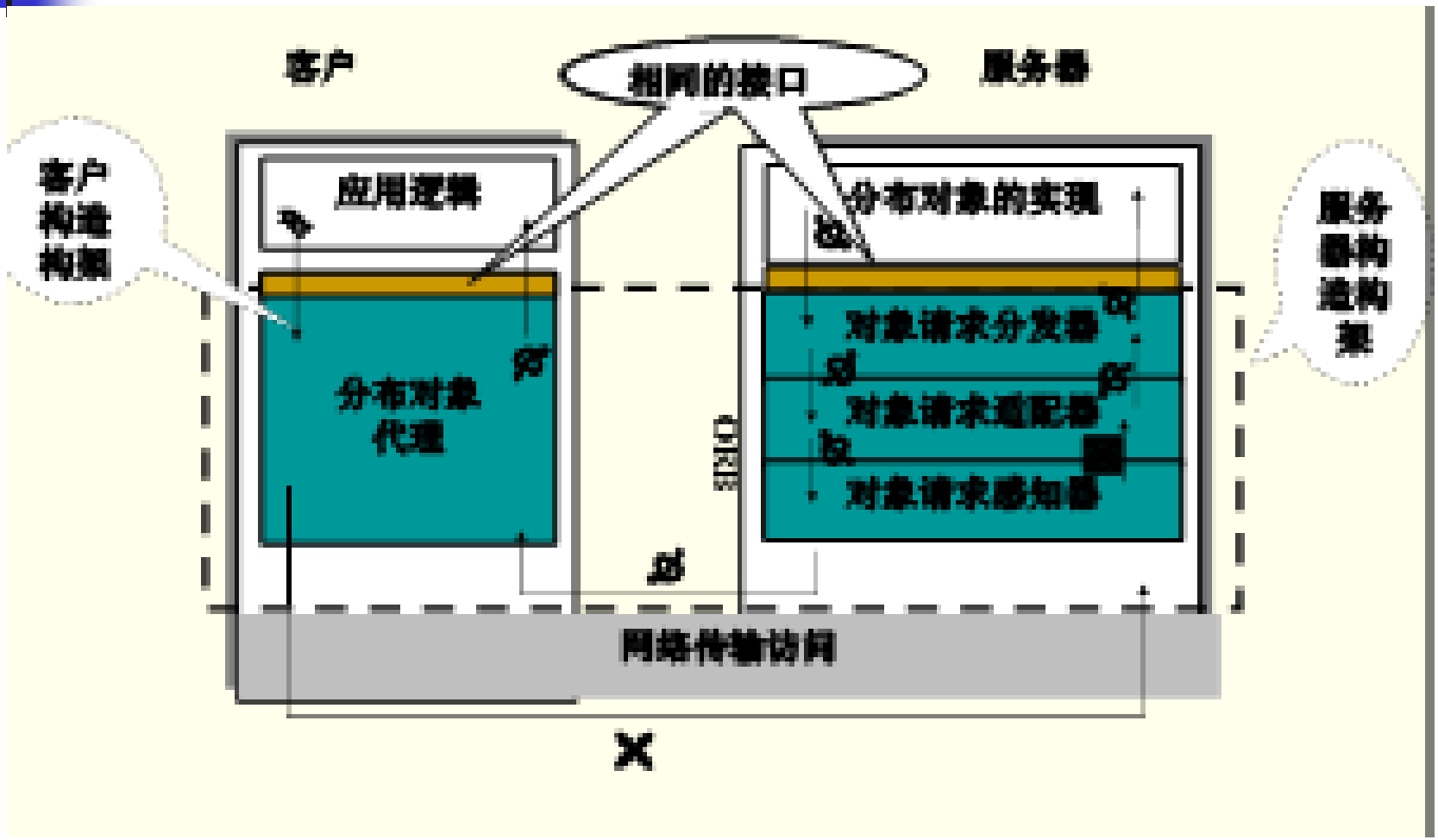


分布对象技术
要解决的基本问题

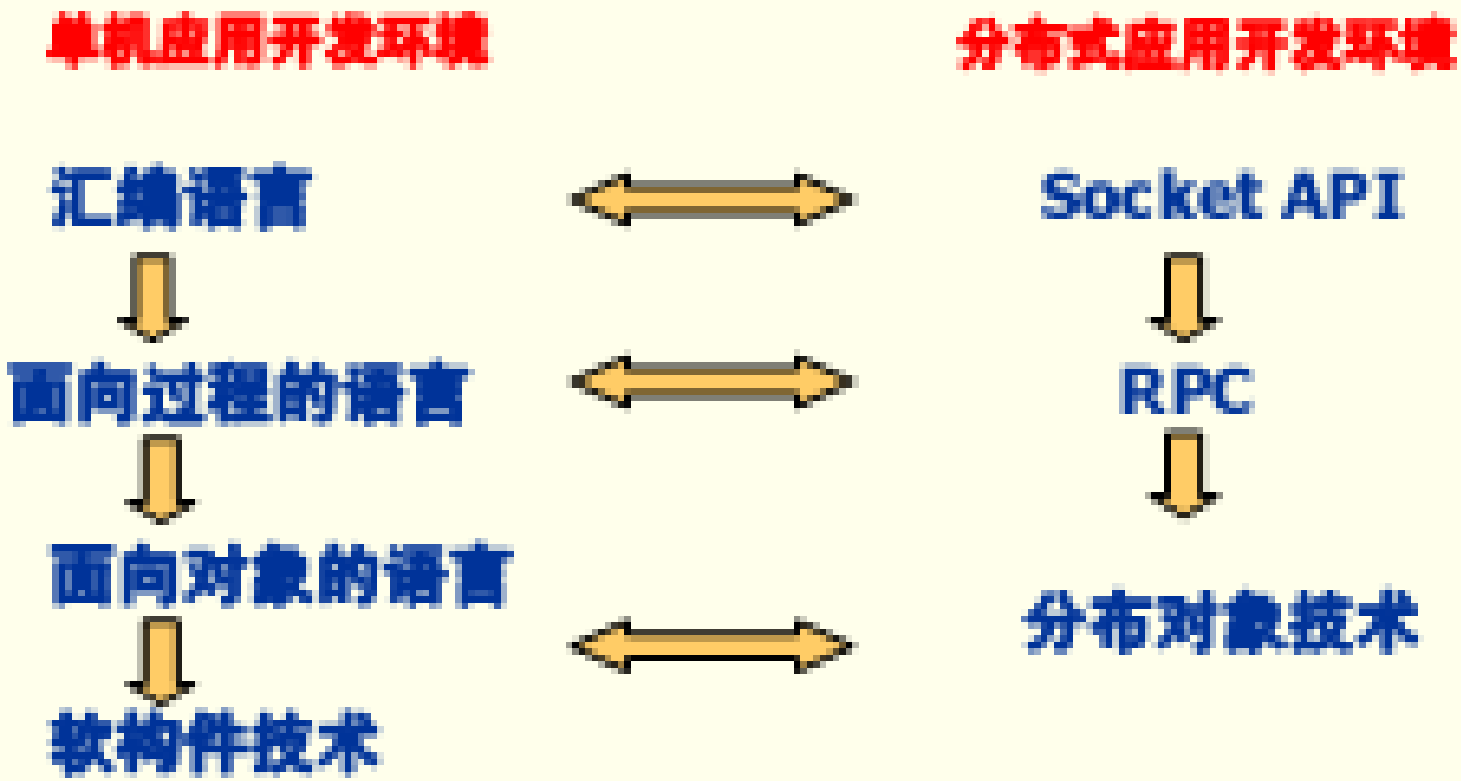
- 对象请求代理---支持客户访问异地分布对象的核心机制称为对象请求代理ORB(Object Request Broker)

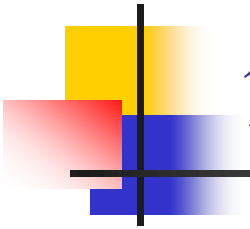


分布对象技术
要解决的基本问题



分布对象技术
要解决的基本问题



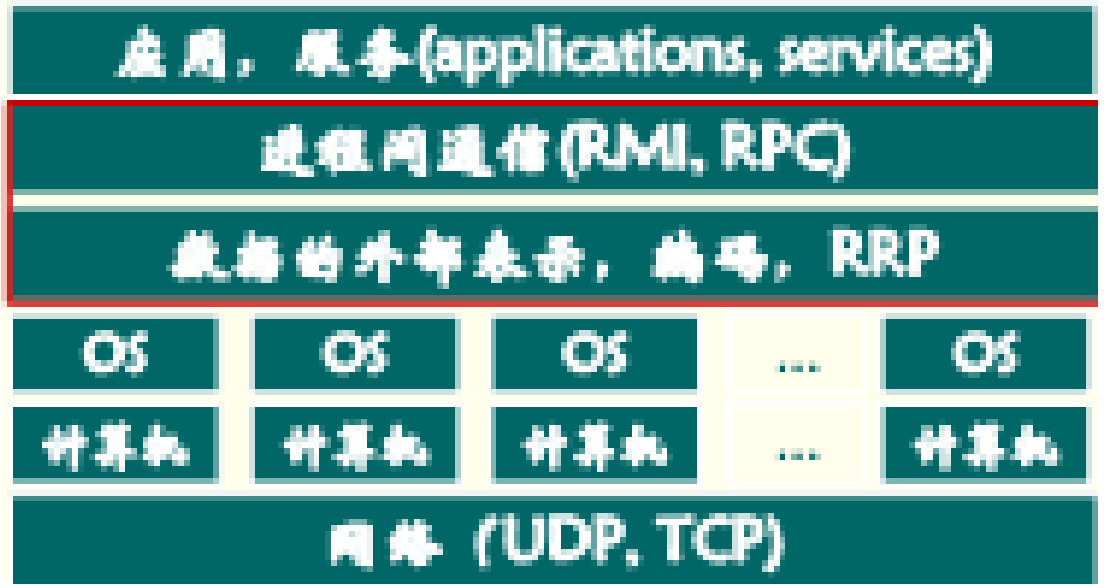


分布式程序设计的模型

- 远程过程调用（RPC--Remote Procedure Call ）
  - 不同进程之间的过程的调用，但是是相同的语言 环境。
- 远程方法调用（RMI--Remote Method Invocation ）
  - 一个进程调用另一个进程中对象的方法，两个进 程可以在同一台主机，也可以在不同的主机。
- 事件驱动（Event-based model）
  - 注册一些感兴趣的对象的事件
  - 事件发生时会的到通知

中间件

- 中间件层
  - 中间件是一种软件，它提供基本的通信模块和其他一些基础服务模块，使得应用程序开发提供平台



中间件要解决的问题

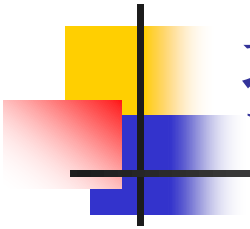
- 通信协议：
  - 独立于网络底层的传输协议。
- 硬件：
  - 数据类型在不同的硬件平台上有不同的表示：big endian，little endian，可以通过mashell解决。
- 操作系统：
  - 在操作系统层上提供更高级的抽象API，屏蔽操作系统的异构
- 编程语言：
  - CORBA通过IDL，可以使得不同的语言写的代码互相调用。
- 中间件技术提供了一个编程的抽象，来屏蔽上述的异质问题。

中间件---分布对象的核心技术

- 中间件的特性
  - 位置透明（Location transparency）
  - 通信协议（Communication protocols）
  - 对计算机硬件层的支持
  - 对操作系统层的支持
  - 对程序设计语言层的支持

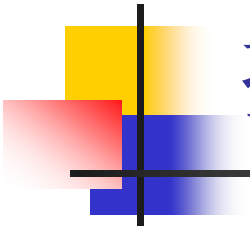
接口（Interfaces）

- 接口
  - 一个模块的接口包含了其他模块可访问的方法的定义（没有实现）和变量。
  - 有些情况，需要调用同一个方法，但是具体的实现的不同，接口可以做到这一点。
- 分布式系统中的接口
  - 同一个地址空间，模块之间的通信可以通过访问公共变量，但是远程调用不能直接访问变量
  - 只能通过输入参数和输出参数
  - 指针不能作为参数传递或者作为结果返回



接口的实例

- RPC's Service interface
  - 对服务器一组过程的说明，定义每个过程的输入输出参数，供客户端调用。
- RMI's Remote interface
  - 对一个对象的方法的说明。
  - 可以传递一个对象或者远程对象的指针，也可以返回这两种类型，这是与过程调用最大的不同。

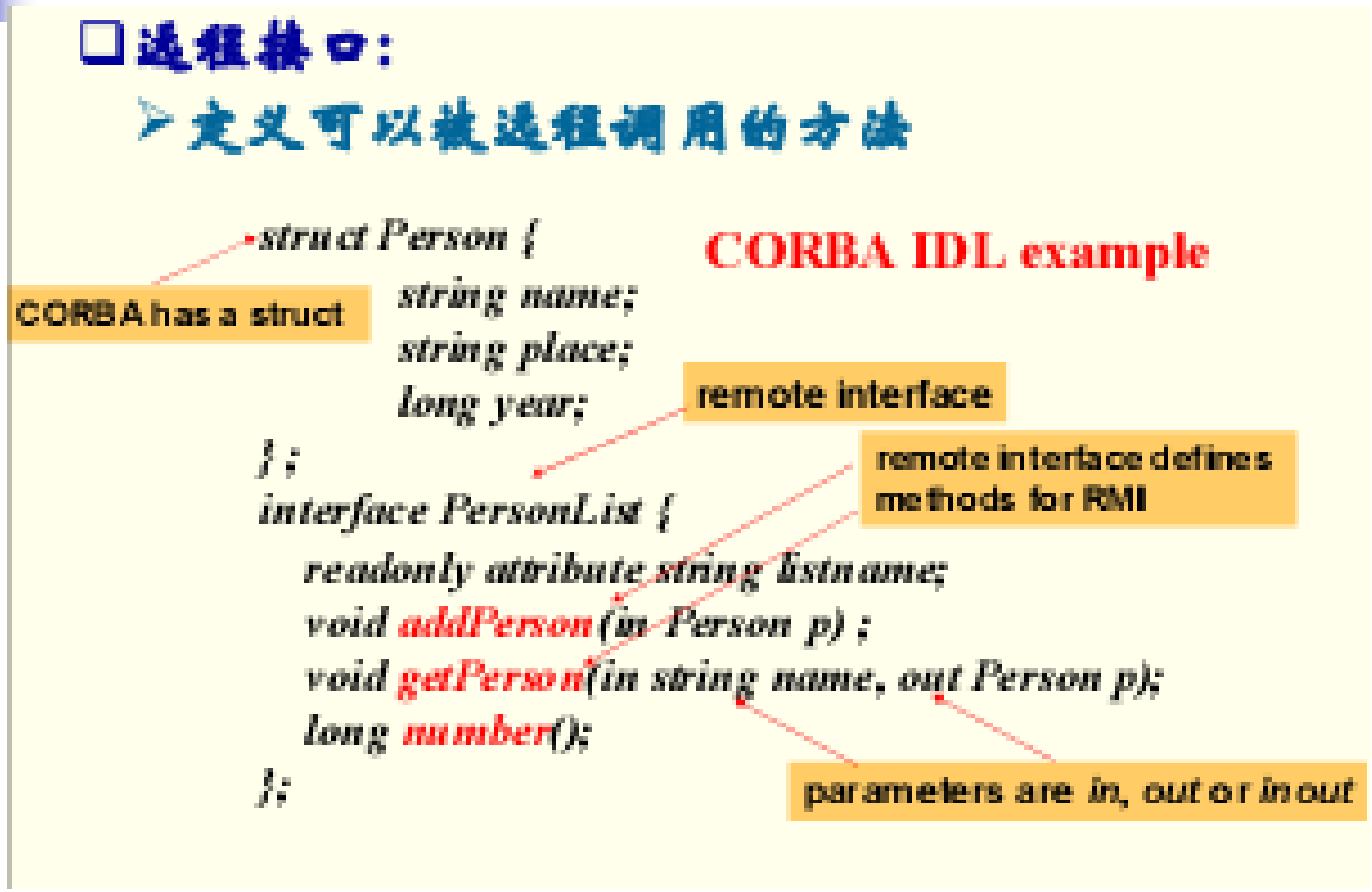


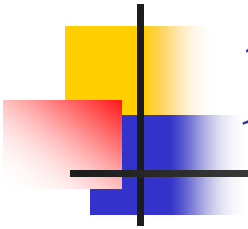
接口的实例

- 接口定义语言（IDLs—Interface Definition Languages ）
  - Java RMI中可以直接定义接口，只能被java语言调用
  - 其它中间件系统提供了IDLs. e.g. CORBA IDL (n1)， DCE IDL and DCOM IDL
  - 允许用其它语言实现的对象来调用，具有跨平台功能

接口的实例

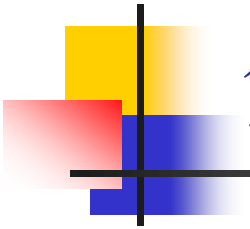
- 远程接口：
  - 定义可以被远程调用的方法





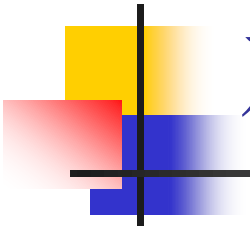
第4章:分布式对象和远程调用

- 引言
- 分布式对象间的通信
- 远程过程调用Remote procedure call
- 事件与通知Events and notifications
- Java RMI 实例研究
- 总结



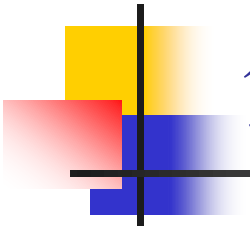
分布式对象间的通信

- 对象模型（The object model）
- 分布式对象（Distributed objects）
- 分布式对象模型（The distributed object model）
- 设计问题
- 实现
- 分布式垃圾回收



对象模型

- 对象的引用（object reference）
  - 访问对象，必须知道对象的引用。可以作为参数的传递，也可以作为结果返回
- 接口（interface）
- 动作（Actions）
  - 一个对象调用了另一个对象的方法则触发一个动作，两个作用：
    - 对象的状态发生改变
    - 生成一个新的对象
    - 连锁的调用
- 异常（Exceptions）
- 垃圾回收（Garbage collection）

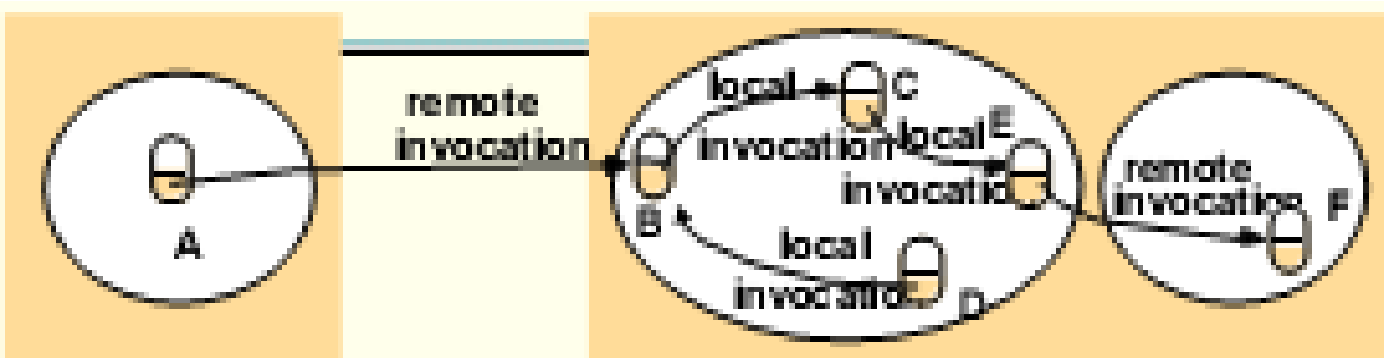


分布式对象

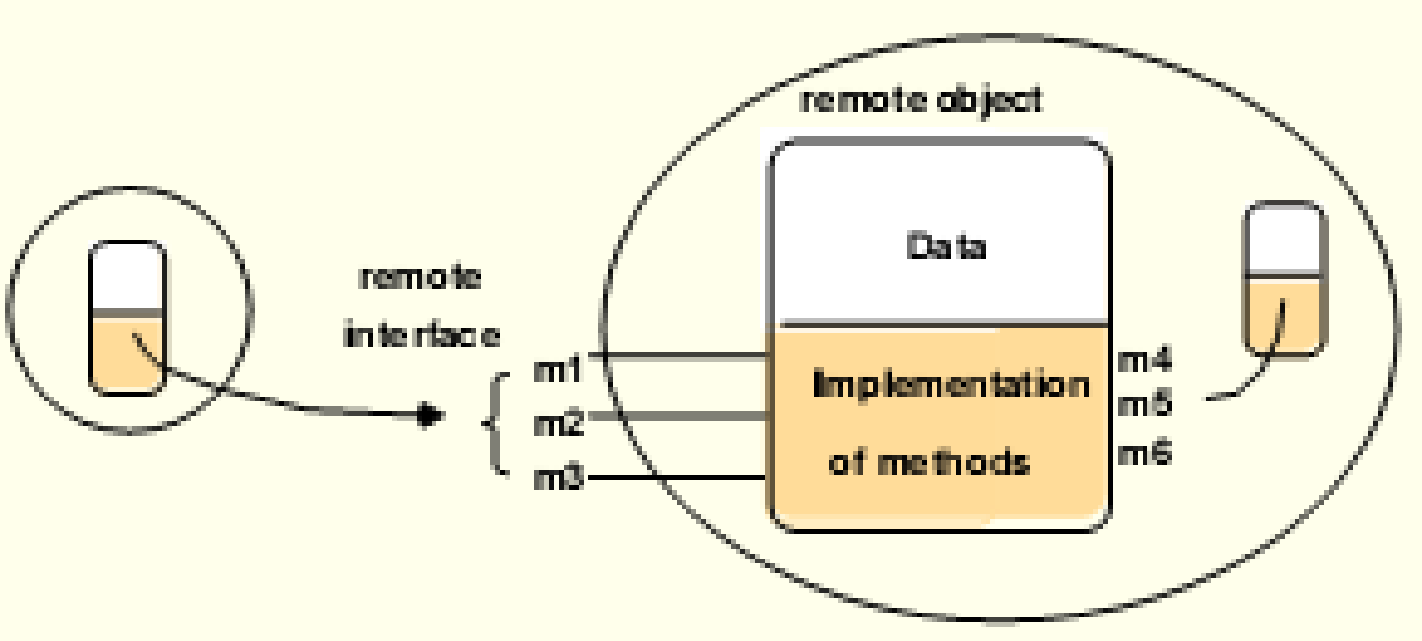
- 除了对象的属性，分布式对象系统的关键是对象之间的通信
- 引入服务器和客户端的概念
  - 客户/服务器模式，对象相当于server，调用该对象方法的程序是client。作为server的对象需要访问其它对象的方法时，它就是client。

分布式对象模型

- 进程中可以包含多个对象，本地的和远程的。调用其它进程的对象叫远程调用，即使在同一台机器上，不同的地址空间的调用也叫远程调用.
- 提供可被远程调用方法的对象叫远程对象。 E.g. B & F
- 调用远程对象必须先获得远程对象的引用。E.g. C 必须有E的引用。
- 远程对象通过远程接口定义可被远程访问的方法。E.g., B和F必须公布远程接口。



分布式对象模型


remote object
Data
remote
interface
m1
m2
m3
Implementation
of methods
m4
m5
m6

分布式对象调用的语义（Invocation semantics）

- 本地调用一定会执行，且只执行一次。
  - exactly once
- 远程调用不能保证exactly once语义，远程调用使用RRP协议，它有三种语义形式：
  - Maybe –客户端没有接收到回复，客户端能判断的结论是：请求可能被执行，返回结果丢失；也可能没有被执行，请求消息丢失或者远程对象的机器不宕机。
  - 如果在设计上没有考虑重发请求，则是maybe语义
  - 存在的问题：不能保证系统正确工作。

分布式对象调用的语义（Invocation semantics）

- At-least-once – 客户端得到一个结果，说明请求至少被执行了一次，也可能多于一次。在它收到一个异常时，应该重发请求，直到得到结果。（Sun RPC）

- 存在问题：
  - 任意故障：如果调用请求被重复发送，远程方法可能被执行多次，多次执行可能引起错误的结果。如果对象提供的操作是幂等操作， 就不存在任意故障。
  - 增加消息量：当远程服务器不工作，会导致不断重发请求，增加消息量。可以采用超时判断，来结束重发。

分布式对象调用的语义（Invocation semantics）

- At-most-once – 客户端收到一个结果，表明请求被执行了一次，并且只执行一次；或者收到一个异常，表明方法没有被执行到（JavaRMI）。
  - 有些应用不允许一次请求被执行多次，需要这种语义。
  - 措施：在服务器端过滤重复请求，或者在在服务保存结果，可以保证方法不被重复执行。
- 存在问题：增加服务器处理的负担。

分布式对象调用的语义（重传策略）

- 分布对象的远程调用不能保证每次调用都确定被执行。
- RRP协议通常采用一些措施来保证所要求的消息传递保障。
- 对于RRP协议，选择不同的策略，以提供不同的提交保证。
  - Retry Request message:是否重发请求，直到收到应答或者确信服务器已经不工作了。
  - Duplicate filtering:当有重复传输的情况下，是否在服务器端过滤掉重复的请求
  - Retransmission of result:是否保存结果的历史记录用于重发，以便在结果丢失的情况下 不需要重复执行请求

分布式对象调用的语义（Invocation semantics）

- 根据远程调用的容错手段，决定了调用语义

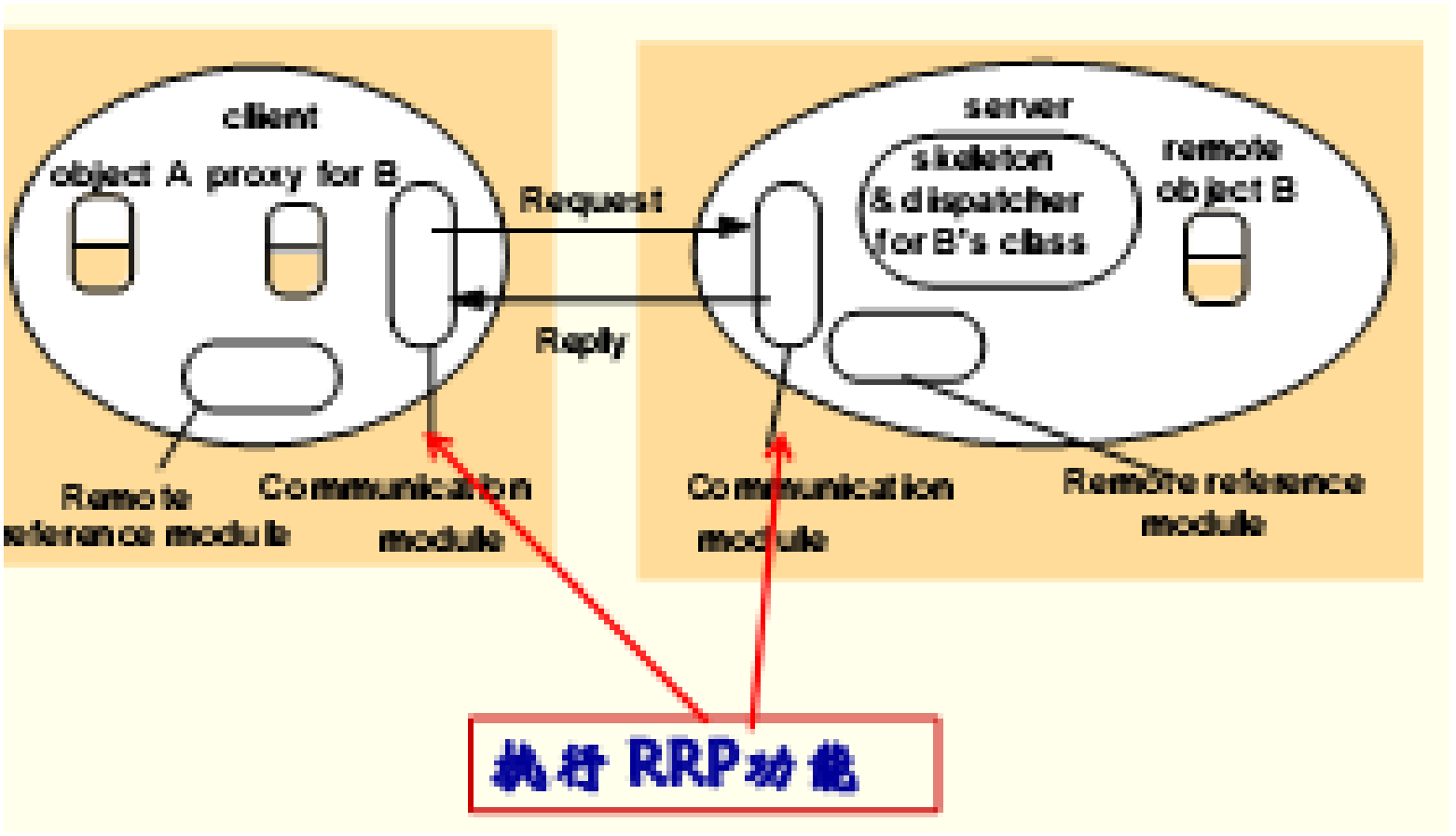
Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

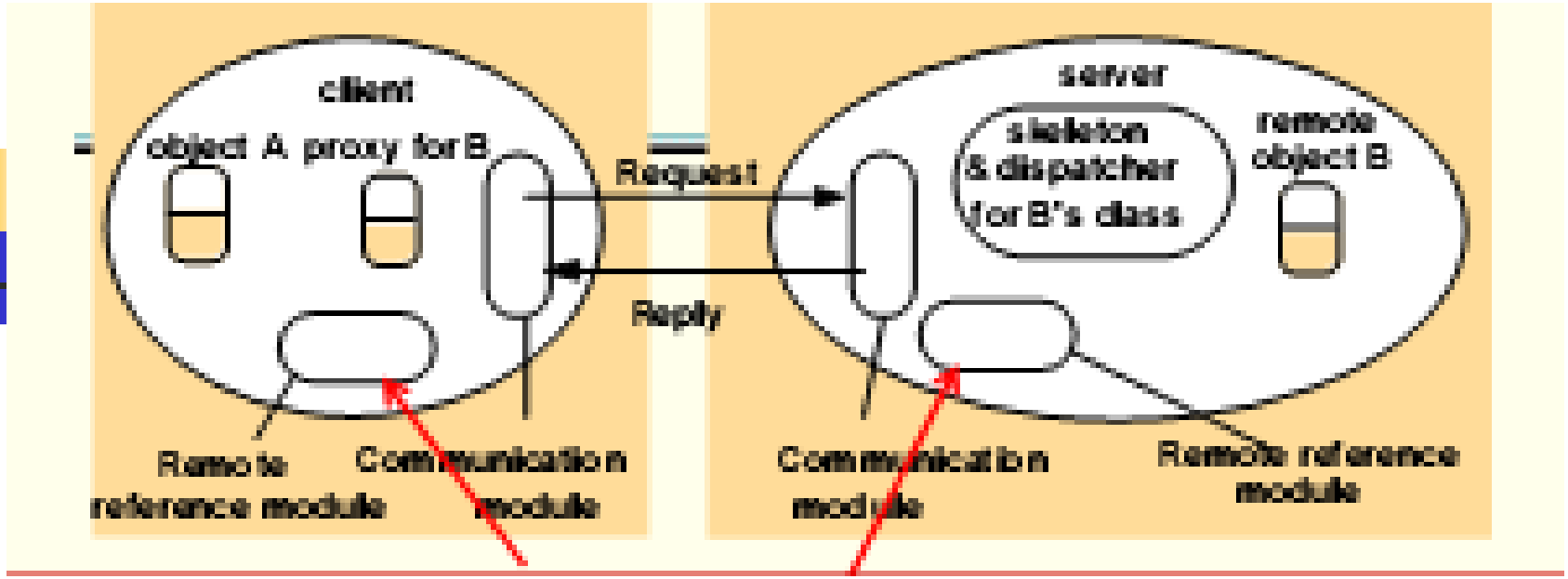
failure model----推断

基于TCP/IP连接的调用语义：

- 当一个进程得知与对方的连接断开了，以下三种情况是可以推断的：
  - 某个进程退出或关闭了连接
    - exactly once.
  - 服务器上的进程崩溃了
    - at most once.
  - 网络拥塞
    - at least once

远程方法调用的结构

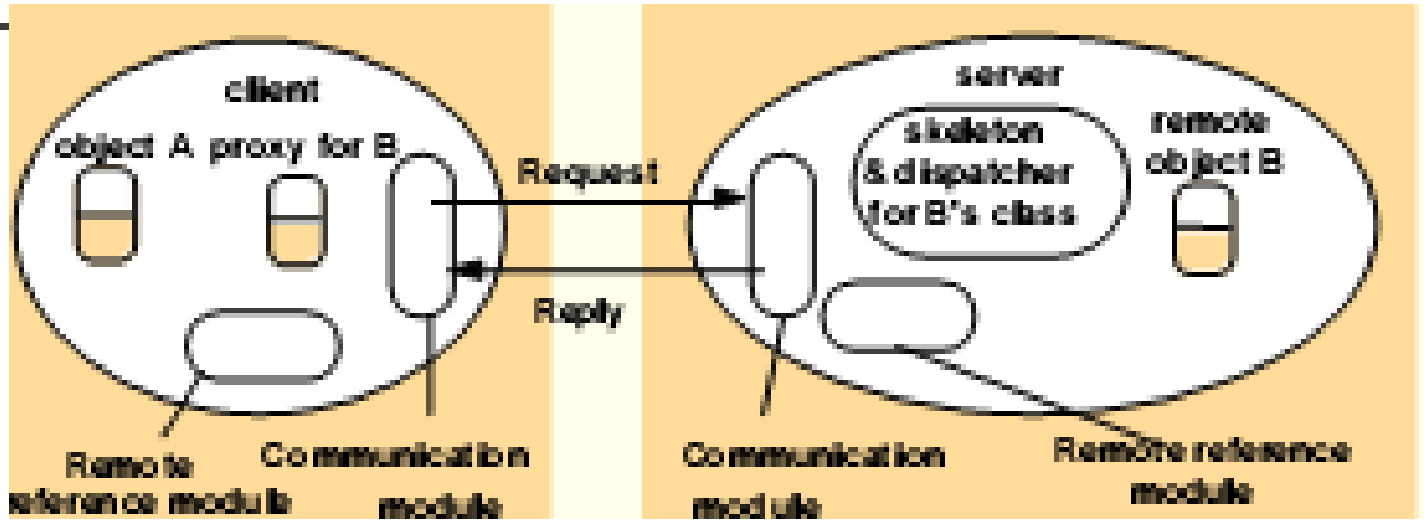

client
object A proxy for B
Request
Reply
server
skeleton
& dispatcher
for B's class
remote
object B
Remote
reference module
Communication
module
Communication
module
Remote reference
module
执行 RRP功能



1、翻译本地对象引用和远程对象引用；2、创建远程对象引用。

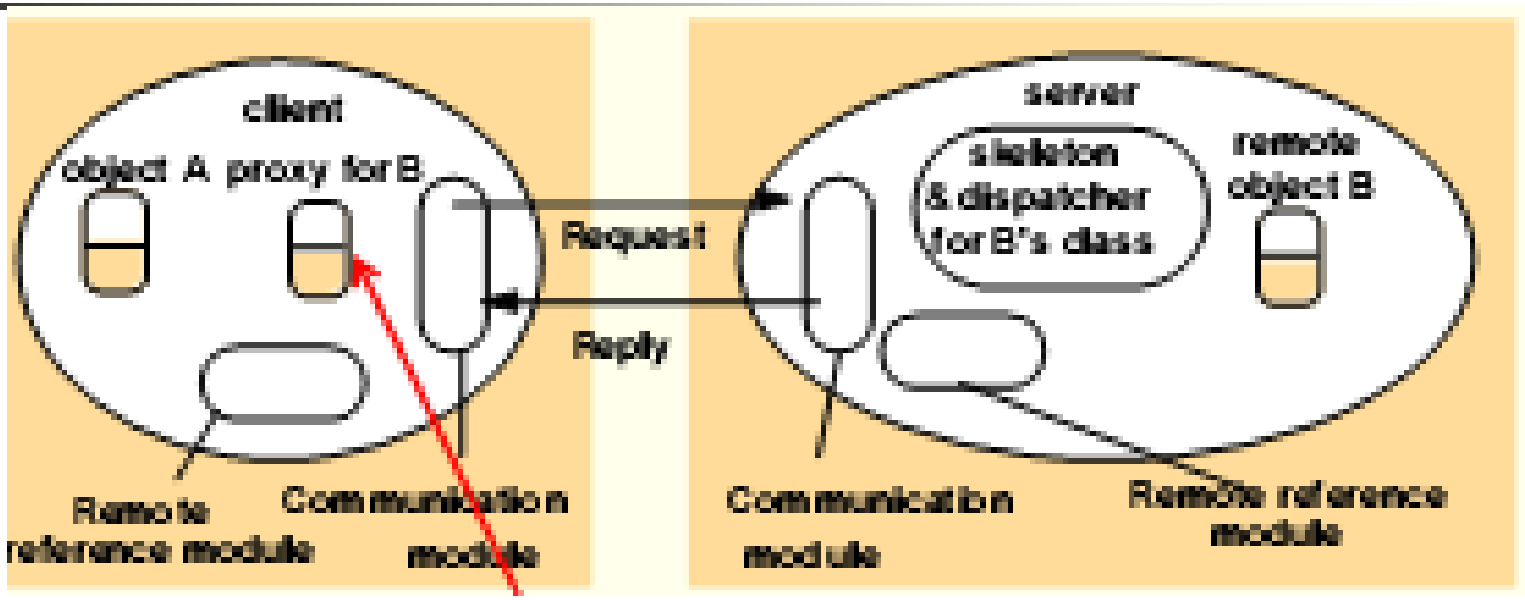
- 每个进程都维护一个远程对象表，记录本地对象引用与远程对象引用的对 应关系。
- 所有远程对象，例如 server的表中要包含B，
- 所有本地对象，例如 client的表中要包含B的代理。
- 当远程对象第一次被传输，RRM为其创建一个远程对象引用，并加到表中
- 当远程对象引用到达接收方，RRM为其建立相应的本地引用，可能指向代 理或者指向远程对象。如果远程对象的引用不在表中，RMI软件就创建一 个新的代理，并由RRM把它加到表中。
- 当软件层进行外部数据的编码解码一个远程引用时，调用RRM，例如，当 一个请求信息到达，表用来查找哪个对象要被调用。

远程方法调用的结构

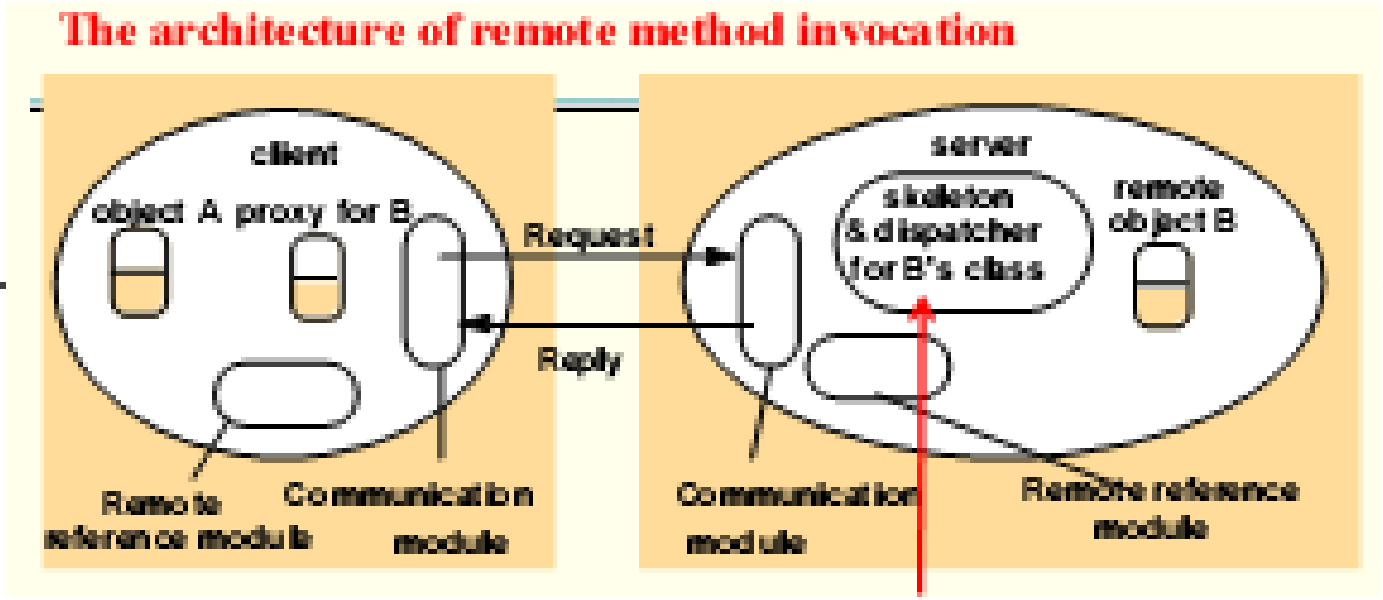


- **RMI software –** 在应用层对象和通讯模块与远程引用模块之间的软件模块。由proxy， skeleton和 dispatcher组成。

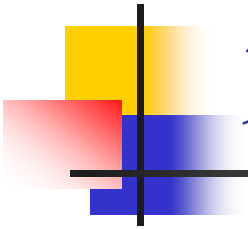
远程方法调用的结构



- proxy的作用是让客户端能透明地调用远程对象的方法。
- 在客户端，每个远程对象都在代理，完成：
  - 将要传输的参数或者对象进行编码解码
  - 传递请求



- 和proxy类似，在server端，一个skeleton和一个dispatcher代理一个远程对象， dispatcher负责接受来自通讯模块发来的请求， 给这个方法。
- Proxy和dispatcher再根据methodId对应方法采用相应的规则。
- 一个远程对象的类有一个skeleton，它实现了远程接口的方法，根据一个对象被调用的用途不同，远程方法的实现也不同
- Skeleton在server端完成编码和解码的功能。



第4章:分布式对象和远程调用

- 引言
- 分布式对象间的通信
- 远程过程调用Remote Procedure Call
- 事件与通知Events and notifications
- Java RMI 实例研究
- 总结

Sun RPC case study

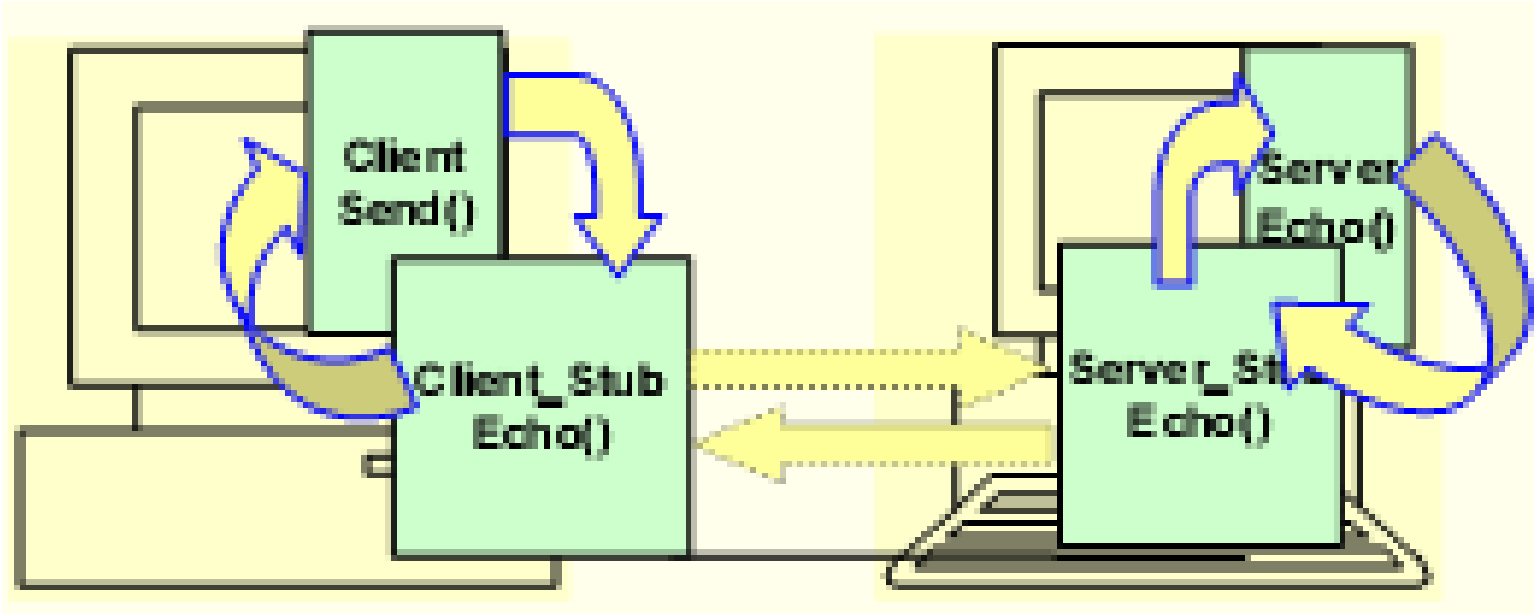
- RPC的目的是可以像程序调用一样使用远程服务。RPC 使用的是客户机/服务器模式。
  - 服务程序就是一个服务器(server)，
    - server提供一个或多个远程过程；
  - 请求程序就是一个客户机(client)，
    - client向server发出远程调用

Sun RPC case study …*continued*

- 程序调用：如何使远程的过程调用看起来和本地的过程调用没有区别的问题
- 网络通信：调用进程和被调用进程间的网络比本地计算机有更复杂的特性。例如，它可能限制消息尺寸，并且有丢失和重排消息的可能，安全问题；
- 操作系统：运行调用和被调用进程的计算机可能有明显不同的体系结构和数据表示格式；
- 编程语言：跨语言之间的互操作问题；

Sun RPC case study …*continued*

- RPC引入了存根（Stub）的概念



Sun RPC case study …*continued*

- 比如服务端有某个函数fn()，它为了能够被远程调用，需要通过编译器生成两个stub：
  - 客户端的一个stub：c_fn()
  - 服务器端的一个stub：s_fn()
- 在客户端，一个进程在执行过程中调用到了函数fn()，此函数的具体实现是在远程的某台机器上，那么
  - 此进程实际上是调用了位于当地机器上的另外一个版本的fn()（即c_fn()）
  - 当客户端的消息发送到服务器端时，服务器端也不是把消息直接就交给真正的fn()，而是同样先交给一个不同版本的fn()（即s_fn()）

Sun RPC case study …*continued*

- Stub的主要功能是对要发送的参数进行marshal(可理解成一种“打包”操作)和对接受到的参数（或返回值）进行unmarshal(解包)。
  - Marshal操作将要发送的数据制成一种标准的格式（在DCE RPC系统中，此格式称做Network Data Representation（NDR）格式）
  - unmarshal再从NDR格式数据包中读出所需数据。

Sun RPC case study …*continued*

- Client stub的功能：
  - 收集调用远程函数需要的参数
  - 将这些参数marshal成消息，即把消息转化成标准的网络数据表示(network data representation，NDR)格式，用于在网络上传递
  - 调用客户端的运行时系统（Client runtime system）将此消息发送给服务器端。
  - 当服务器端将结果消息返回后，将结果消息unmarshal，把结果返回给应用进程。

Sun RPC case study …*continued*

- Server stub的功能：
  - 对发送给它的参数消息unmarshal，收集参数
  - 调用位于本机上的过程
  - 将此过程执行的结果marshal成消息，然后调用服务器端的运行时系统将结果消息发送给客户端

Sun RPC case study …*continued*

- IDL compiler的功能就是对编辑好的IDL文件进行编译，编译后生成了下面的三个文件：
  - Header是一个头文件，此头文件包含了此IDL文件中的全局唯一标示符，数据类型定义，有关的常量定义，以及函数原型。客户代码和服务器代码中包含都要包含header文件
  - Client stub即客户端的stub程序。
  - Server stub即服务器端的stub程序。

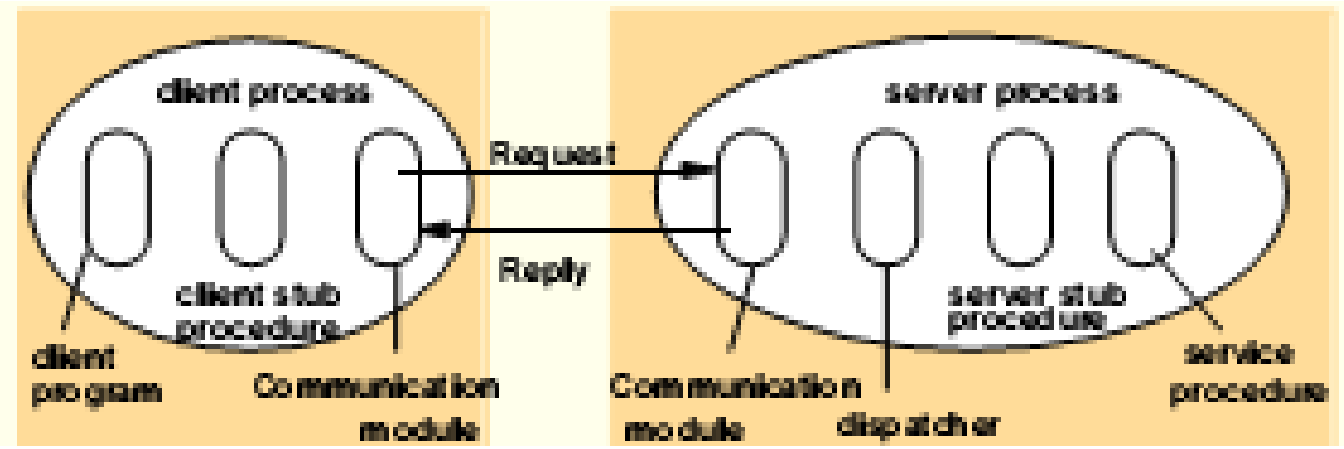
Sun RPC case study …*continued*

- Client code和Server code就是由应用程序开发人员所写的客户代码和服务器代码，客户代码可能调用到各种各样的过程，服务器代码就是这些过程的真正实现。
  - 客户代码和客户stub代码分别经过编译生成各自的目标文件，这些目标文件再与运行时库连接就生成了整个客户端可执行文件。服务器端也同理。

Sun RPC case study …*continued*

- 运行时系统
  - 客户端的运行系统将客户端stub产生的消息可靠的传送给server
  - 运行时系统利用TCP/UDP等协议，将消息发送到Server
  - 服务端的运行时系统都侦听某个众所周知的socket端口，接受请求
  - 服务端的运行系统调用Server Stub处理

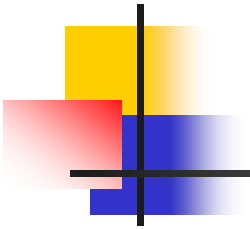
- Service interface: the procedures that are available for remote calling
- Invocation semantics choice: at-least-once or at-most-once
- Generally implemented over request-reply protocol
- Building blocks
  - Communication module
  - Client stub procedure (as proxy in RMI): marshalling, sending, unmarshalling
  - Dispatcher: select one of the server stub procedures
  - Server stub procedure (as skeleton in RMI): unmarshalling, calling, marshalling



Sun RPC case study

- 用于NFS
  - at-least-once semantics
- XDR - Interface definition language
  - Interface name: Program number， version number
  - Procedure identifier: procedure number
- rpcgen – generator of RPC components
  - client stub procedure
  - server main procedure
  - Dispatcher
  - server stub procedure
  - marshalling and unmarshalling procedure

Sun RPC case study …*continued*



- Binding – portmapper
  - Server: register ((program number， version number)， port number)
  - Client: request port number by (program number， version number)
- Authentication
  - Each request contains the credentials of the user， e.g. uid and gid of the user
  - Access control according to the credential information

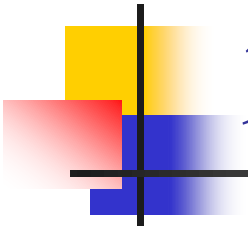
//RPC 调用信息主体形式如下：

```
struct call_body {
    unsigned int rpcvers;
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
    opaque_auth verf;
    parameter 1
    parameter 2. . .
};
```

**Sun Microsystems'
Open Network Computing.
The ONCuses XDR
as the external data
representation standard**

Sun RPC case study …小结

- RPC把通信接口抽象成程序调用，引入Stub，使得调用远程函数看起来就像本地函数调用。
- DCE RPC引入了IDL语言用于描述程序接口，经过编译之后得到stub，保证了接口的一致性。
- RPC系统把参数和返回值编码成用来传输的



第4章:分布式对象和远程调用

- 引言
- 分布式对象间的通信
- 远程过程调用Remote procedure call
- 事件与通知Events and notifications
- Java RMI实例研究
- 总结

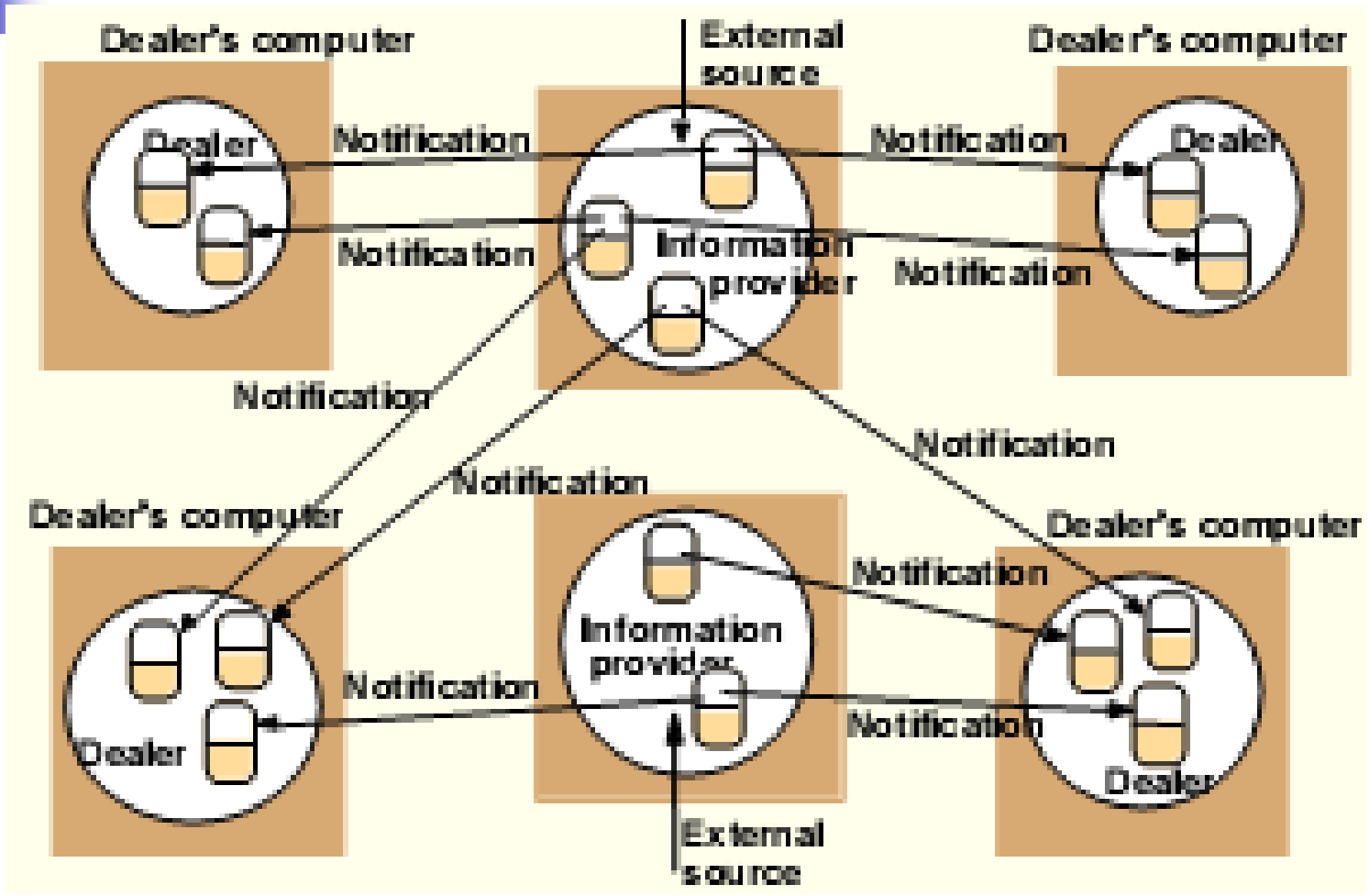
事件与通知模型

- 目的
  - 让一个对象能够对另一个对象发生的变化**做出反应**
- 举例：
  - 文件被修改了
  - 一个电子书签变化了位置
- 发布与订阅（Publish/subscribe paradigm）
  - 产生事件一方发布事件的类型
  - 接收事件一方订阅感兴趣的事件类型
  - 事件放生时，通知订阅一方。
- 分布式事件驱动系统的两个特性：
  - 异构性：已有的分布式系统的一些模块原本不是为了互操作而设计的，现在能够让他们协同工作，通过接收方公布远程接口。
  - 异步性：不能要求发布程序和订阅程序同步。

事件与通知模型

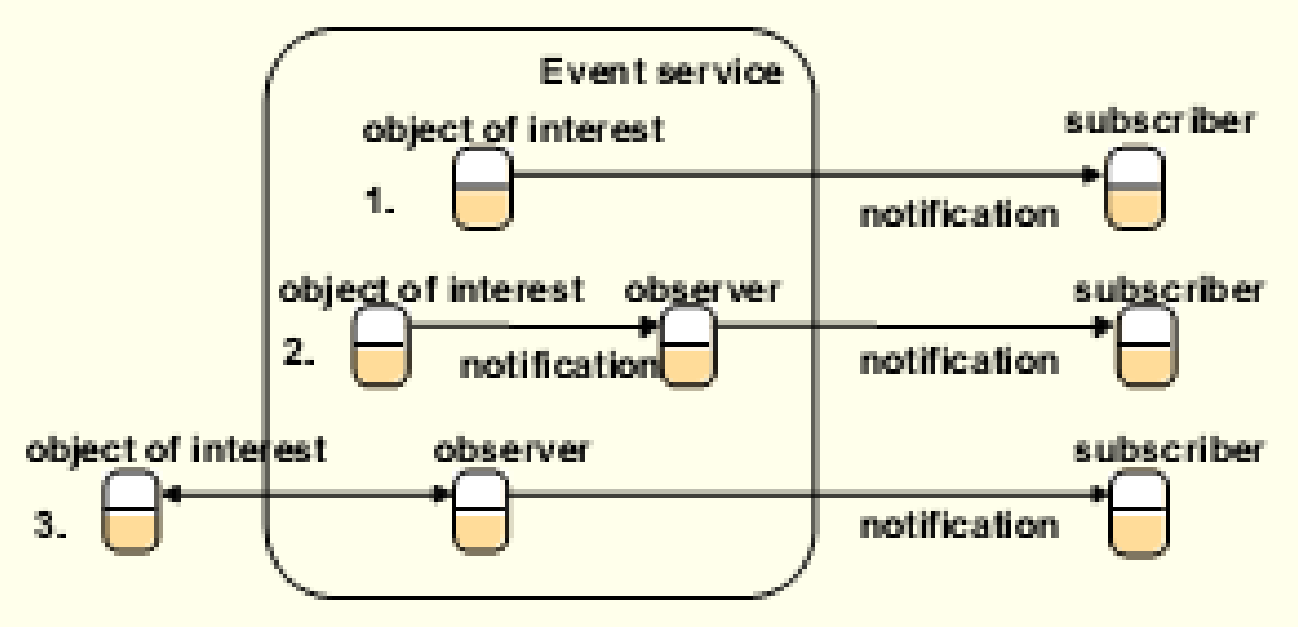
- 举例：交易所系统
- 任务：
  - 让交易者能够查看他们交易的股票的最新市场行情。
- 系统构成
  - 信息提供者（Information provider）
    - 接收新的交易信息
    - 发布股票价格
    - 股票价格变化通知
- 交易进程
  - 订阅股票价格事件

事件与通知模型


Dealer's computer
External source
Dealer's computer
Dealer
Notification
Notification
Dealer
Notification
Information provider
Notification
Notification
Notification
Notification
Dealer's computer
Dealer's computer
Notification
Information provider
Notification
Notification
Dealer
Dealer
External source

事件与通知模型的结构

- 事件服务：维护一个发布的事件类型和订阅者兴趣的数据库，使得双方能够耦合起来。



事件与通知模型的结构

- 兴趣对象
  - 是一个对象，由于其方法被调用而改变自身的状态，这种改变会引起其他对象的兴趣。
- 事件
  - 兴趣对象的方法被执行的结果叫做事件。
- 通知
  - 是一个对象，它包含关于事件的信息（属性，类型等）。
- 订阅者
  - 是一个对象，它订阅另一个对象上发生的事件，并接收通知。
- 观察者对象
  - 任务是将兴趣对象和其订阅者进行耦合.
  - 分担兴趣对象的工作。
- 发布者
  - 是一个对象，声明它产生的一些事件的通知，他可以是兴趣对象，也可以是一个观察者。

事件与通知模型的结构

- 传递语义
- 和进程间消息传递的语义类似，取决于应用需求和所采用的措施。
  - 对可靠性要求较低的需求，一些网络游戏了解其它玩家的最新状态，普通IP组播协议。
  - 对可靠性要求较高的需求，股票交易所系统。需要采用可靠的组播协议。
  - 实时要求。

事件与通知模型的结构

- 观察者的作用
  - 转发
- 代替兴趣对象发送通知
  - 通知过滤
- 减少通知的数量
  - 事件模式
- 订阅者可以定制若干事件之间的关系，满足这些关系时才发通知
  - 通知邮箱
- 异步传输

事件与通知模型的结构

- RSS是一种用于共享新闻和其他Web内容的数据交换规范，起源于网景通讯公司的推"Push"技术，将订户订阅的内容传送给他们的通讯协同格式(Protocol)。RSS可以是以下三个解释的其中一个：
  - Really Simple Syndication （真正简单的整合）
  - RDF (Resource Description Framework) Site Summary
  - Rich Site Summary （丰富站点摘要）
- 这三个解释都是指同一种Syndication的技术。
- RSS目前广泛用于blog、wiki和网上新闻频道，世界多数知名新闻社网站都提供RSS订阅支持。

Jini规范----内容

- 什么是Jini(Genie的谐音)?
  - Jini是Sun公司的研究与开发项目，面向服务的分布式架构
  - Jini技术可使范围广泛的多种硬件和软件---即可与网络相连的任何实体---能够自主联网。
- Jini技术可划分为两个范畴：
  - 体系结构和分布式编程
- 提供在Jini上运行的网络服务基础结构
  - Jini基础结构解决设备和软件如何与网络连接并进行注册等基本问题

Jini规范----组成要素

- 基础结构的第一种要素称作Discovery and Join，它解决设备和应用程序在对网络一无所知的情况下如何向网络进行首次注册这样的难题。
- 基础结构的第二个要素是Lookup(搜索)。Lookup可被看作网络中所有服务的公告板。
- Network Services ---网络服务
- Other Services ---其它服务
- Leasing ---租用
- Transactions ---交易
- Distributed Event---分布式事件
- Other OS ---其它操作系统
- Other CPU ---其它CPU

Jini规范----Discovery and Join

- 设备或应用程序插入网络后需要完成的第一个任务就是
  - 发现该网络
  - 它就通过一个众所周知的端口向网络发送一个512字节的多路广播Discovery包。在
  - 其它信息中，该包包含对自己的引用。

Jini规范----Lookup的作用过程

- Jini Lookup在众所周知的端口上进行监听。当接收到Discovery包后，Lookup就利用该设备的接口将Lookup的接口传递回插接的设备或应用程序。
- 现在，该设备或应用程序已经发现了该网络，并准备将其所有特性上载到Jini Lookup。上载特性是Discovery and Join中Join这方面的特性。
- 现在该设备或应用程序使用在Discovery阶段所接收到的Lookup接口与网络相连。上载到Lookup的特性包括该设备或应用程序所提供的所有增值服务(如驱动程序、帮助向导、属性等)。
- Lookup是网络上所有服务的网络公告板。Lookup不但存储着指向网络上服务的指针，而且还存储着这些服务的代码和/或代码指针。

Jini规范----Lookup举例

- 例如，当打印机向Lookup注册时，打印机将打印机驱动程序或驱动程序接口上载到Lookup。当客户机需要使用打印机时，该驱动程序和驱动程序接口就会从Lookup下载到客户机。这样，就不必事先把驱动程序装载到客户机上。
- 打印机还可能把其它增值服务装载入Lookup。例如，打印机可能存储关于自己的属性(如它是否支持postscript，或它是否为彩色打印机)。打印机还可能存储可在客户机上运行的帮助向导。
- 如果网络上没有Lookup，则网络就会使用一个Peer Lookup(对等Lookup)程序。当需要服务的客户机在网络上找不到Lookup时，Peer Lookup就开始工作。在这种情况下，客户机可发送与Lookup所用的相同的Discovery and Join包，并要求任何服务供应商进行注册。随后，服务供应商就会在客户机上注册.

Jini规范----分布式编程

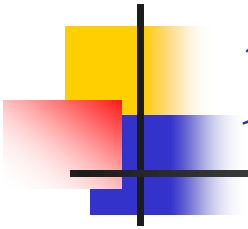
- 分布式编程可提供租用、分布式交易和分布式事件。
- 租用软件与租用一套公寓很类似。我们在租用一套公寓时，一般会商定使用该公寓的时间。类似地，在Jini中，对象彼此之间商定租期。例如，当某设备使用Discovery and Join协议发现网络时，它就注册一段租用时间。在租约到期之前，该设备必须重新商定租期。这样，如果租约到期或设备拔下后，该设备在Lookup中的记录就会被自动删除。这就是分布式垃圾收集的工作原理。

Jini规范----分布式提交

- 分布式交易在分布式Java环境中，有时需要一种很简便的方法，来确保在整个交易完成之前，在该交易中发生的所有事件都被真正提交了（两阶段提交）。
- 为便于进行此类分布式计算，Jini提供了一种简单的Java API。该API可使对象起动一个能管理交易的交易管理器。每个参与交易的对象都向交易管理器注册。
- 当交易发生时，如果某个参与的对象说，交易中的某个事件没有发生，则此信息就被送回交易管理器。随后，交易管理器就告诉所有参与的对象回滚（rool back）到前一个已知状态。类似地，如果所有对象都完成了其交易的过程，则整个交易就向前进行。
- Jini上的网络服务在Jini基础结构和分布式编程之上，可提供便于分布式计算的网络服务。JavaSpace就是这样的一种网络服务。

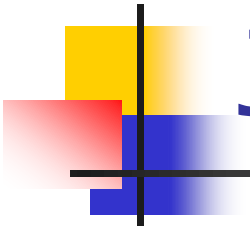
Jini规范----分布式事件

- 分布式事件在单一的计算机中，事件肯定能被接收方接收到，序列也肯定能按照顺序进行。
- 但在分布式环境中，分布的事件可能不是按照顺序被接收，或者，某个事件还可能丢失。
- 为便于在Java环境中处理分布的事件，Jini为分布的事件提供了一个简单的Java API。例如，当一个分布的事件发生时，该事件都带有一个事件号和序列号。利用这种信息，接收方就能检查事件是否丢失(序列号丢失)或事件是否按照顺序接收(序列号顺序不对)到。



第4章:分布式对象和远程调用

- 引言
- 分布式对象间的通信
- 远程过程调用Remote procedure call
- 事件与通知Events and notifications
- Java RMI实例研究
- 总结

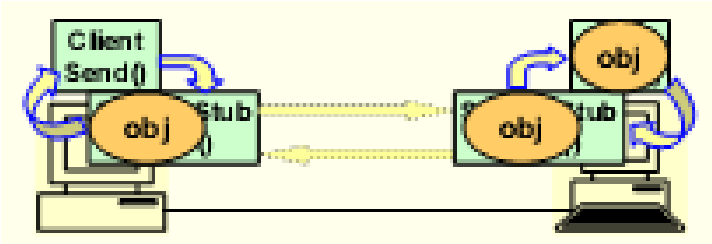


Java RMI实例研究

- remote method invocation(RMI)
- 出现于Jdk1.1(1997.02)，在Jdk1.2中改进
- 定位：
  - Access to Remote Objects
  - Client-Server Protocol
  - High-level API
  - Java-to-Java only
  - Transparent
  - Lightweight

Java RMI 实例研究----对比RPC

- 从RPC到RMI
- 变化：
  - 从过程（静态）->面向对象（动态）
  - 从函数调用->对实例的操作
- 问题：
  - 谁来创建实例对象？什么时候创建？
    如何取得对象的引用？



Java RMI实例研究----组成

- RMI系统由以下几个部分组成：
  - 运行远程服务的服务器
  - 需要远程服务的客户端程序
  - 远程服务的接口定义(Remote Interface)
  - 远程服务的实现(Remote Service)
  - Stub和Skeleton文件
  - RMI命名服务，使得客户端可以发现远程服务

Java RMI实例研究---- RMI编程

- 编写并编译接口的Java代码
- 编写并编译实现类的Java代码
- 利用RMIC从实现类产生Stub和Skeleton类文件
- 启动注册服务
- 编写远程服务主机(host)程序的Java代码，运行之
- 开发RMI客户端程序的Java代码，运行之

Java RMI实例研究----组成

- RMI应用通常由两部分组成：server，client
  - 服务程序：创建远程对象，建立这些对象的引用，等待来自客户端的调用
  - 客户端程序：取得远程对象的远程引用，调用其方法.
- RMI提供了server和client信息传递的机制
- RMI的这种应用被称为分布式对象应用

Java RMI实例研究---定义接口

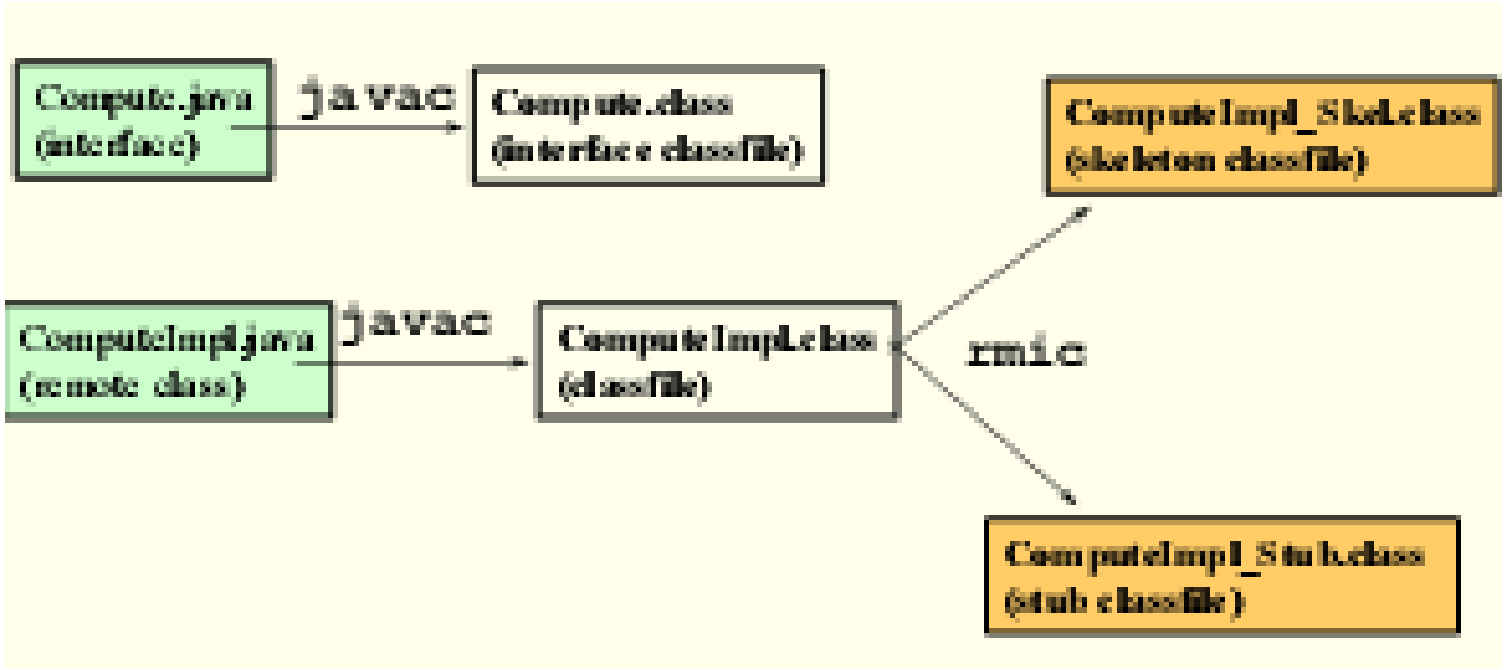
远程接口：

```
import java.rmi.*;
public interface compute extends Remote
   {
public int add(int x，  int y)
throws RemoteException;
}
```

Java RMI实例研究---开发远程对象

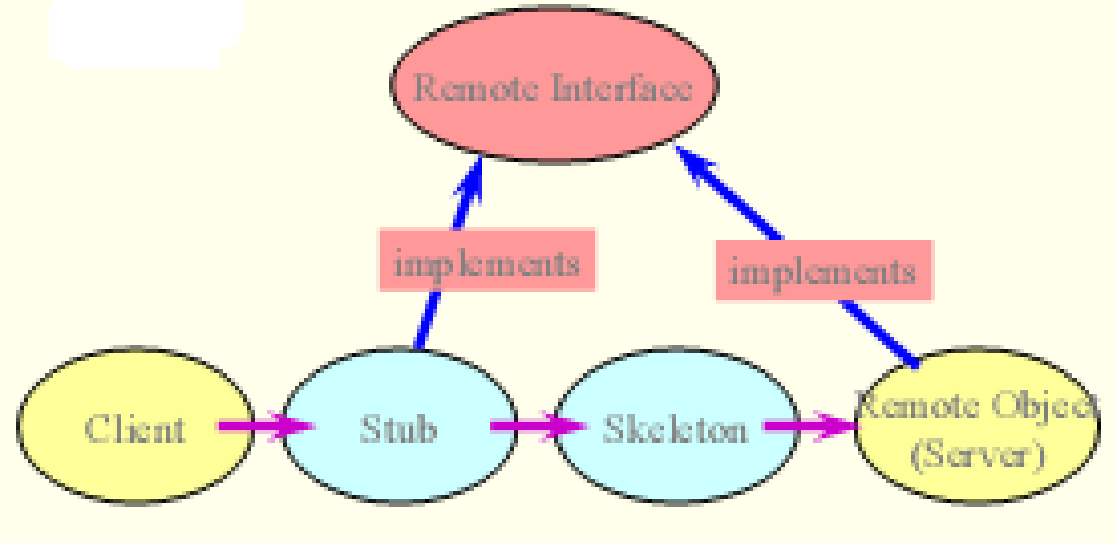
```
import java.rmi.*;
import java.rmi.server.*;
public class AdderImpl extends UnicastRemoteObject
implements Compute {
public AdderImpl()  throws RemoteException {
}
public int add(int x，  int y)
throws RemoteException  {
return x + y;
}
```

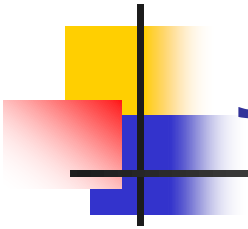
Java RMI实例研究---编译远程类



Java RMI实例研究---生成stubs和skeleton

- Stubs和skeletons利用rmic编译器产生
- Stub和skeleton类是在运行时确定，并动态加载的





Java RMI实例研究---参数的类型

- 基本类型
  - 值传递
- 远程对象
  - 传递引用
- 非远程对象
  - 值传递
  - 对象序列化（Object Serialization）
    - 把对象写成字节序列
    - 写入流(Stream)
    - 在另一端重建
    - 根据旧的数据创建一个全新的对象

Java RMI 实例研究---注册

- 命名和查找远程对象
- 服务器可以注册它们的对象
- 客户端可以发现服务对象并且获取远程引用
- 注册器是运行在主机上的进程

Java RMI 实例研究---注册

- RMI包含了一个简单的目录服务，称为RMI Registry， 它
  - 运行在每个有远程服务对象的主机上
  - 接受服务查询请求，缺省端口是1099.(object servers will be dynamically assigned ports by the RMI runtime)
  - 支持stub代码的动态下载

Java RMI 实例研究---注册

- Server-side
  - 创建一个实现远程服务的本地对象
  - 导出该对象给RMI (创建一个侦听服务，等待客户端的连接).
  - 用一个公开的名称，将该对象注册到 RMI Registry中

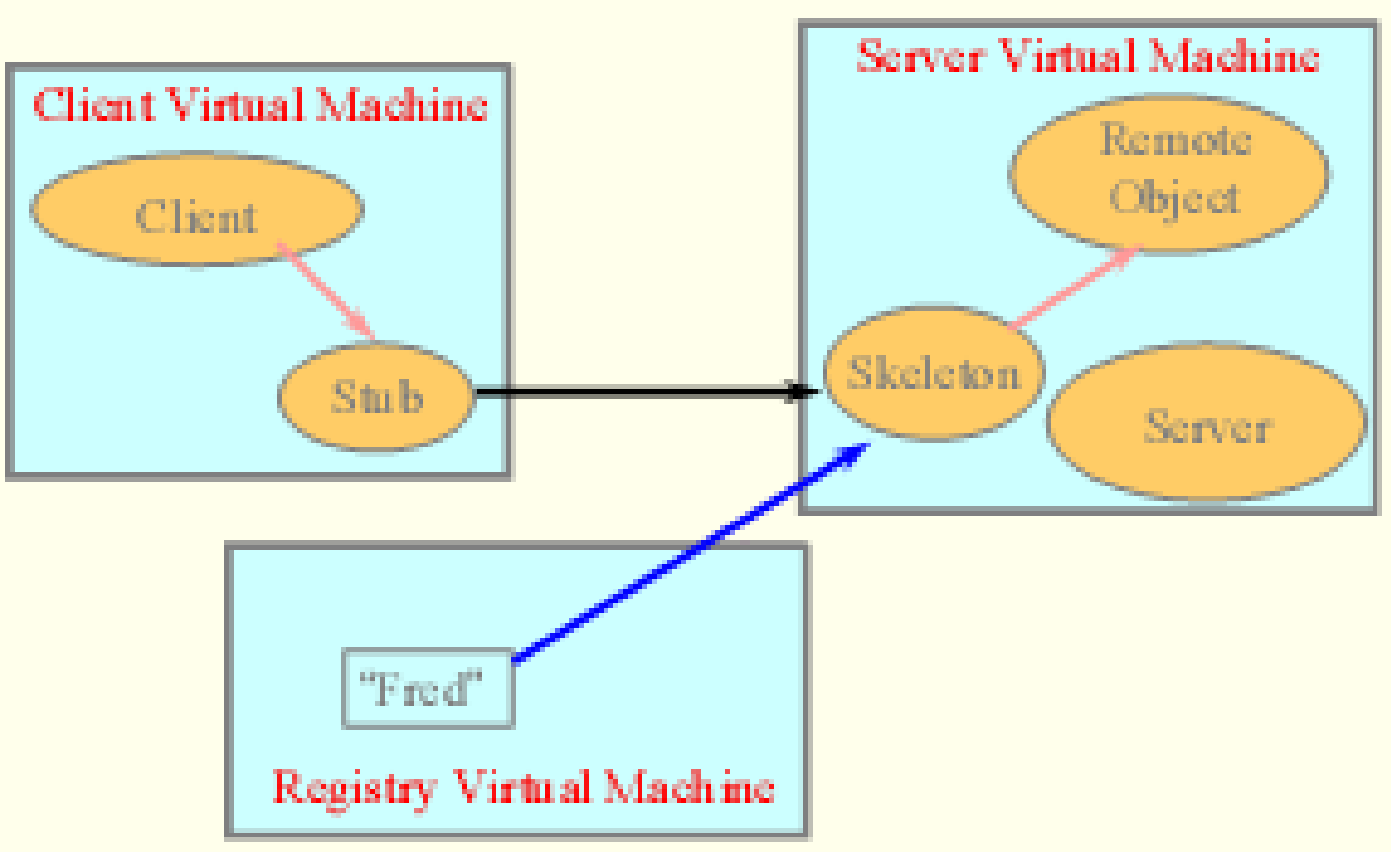
Java RMI 实例研究---注册

- Client-side
  - 通过java.rmi.Naming 静态类访问RMI Registry
  - 使用方法lookup()查询注册器，参数是URL:

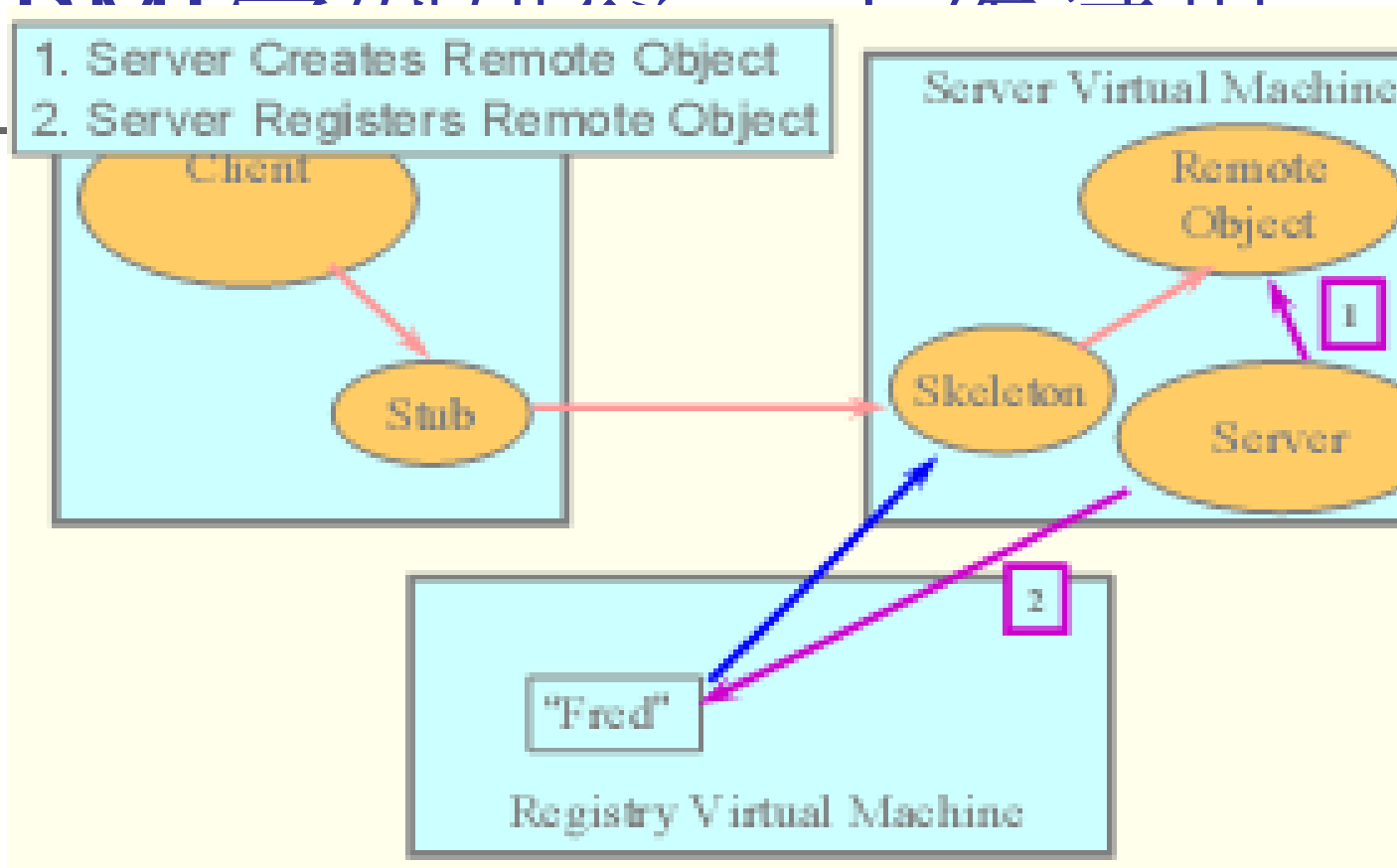
rmi://<host_name>[:<name_service_port>]/<service_name>

该方法返回服务对象的远程引用

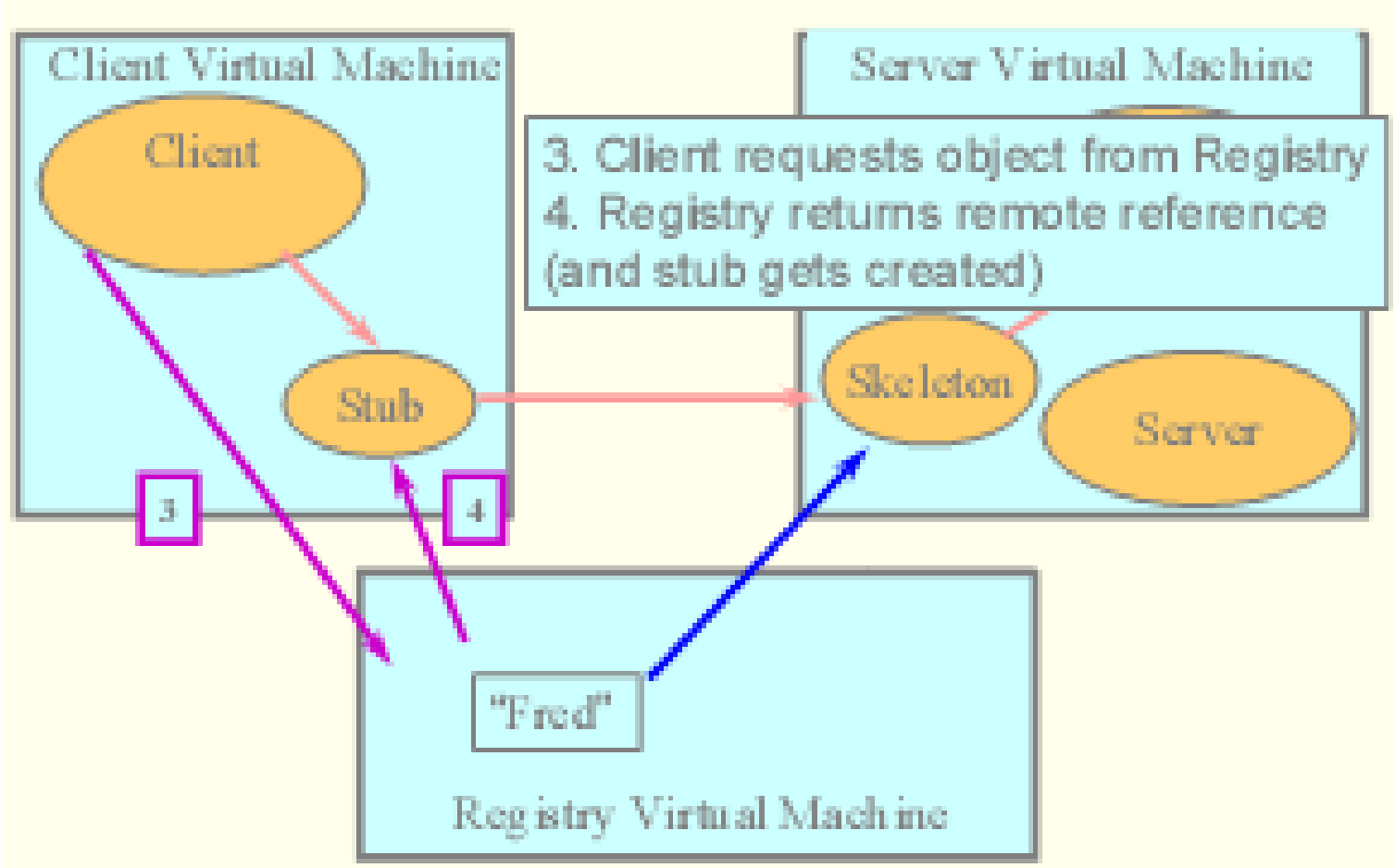
Java RMI实例研究---工作过程



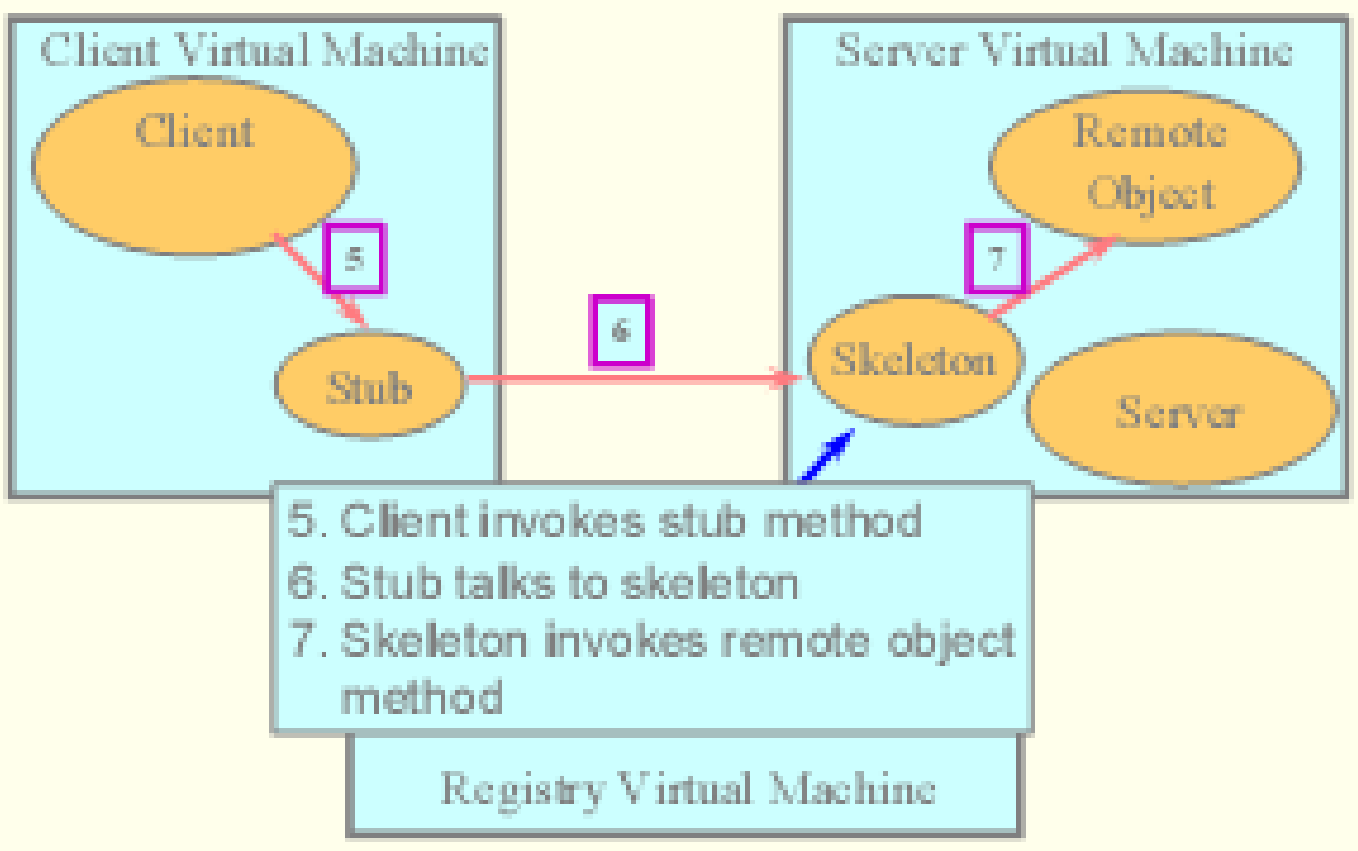
Java


1. Server Creates Remote Object
2. Server Registers Remote Object
Client
Stub
Server Virtual Machine
Remote Object
1
Skeleton
Server
2
"Fred"
Registry Virtual Machine

Java RMI实例研究---工作过程

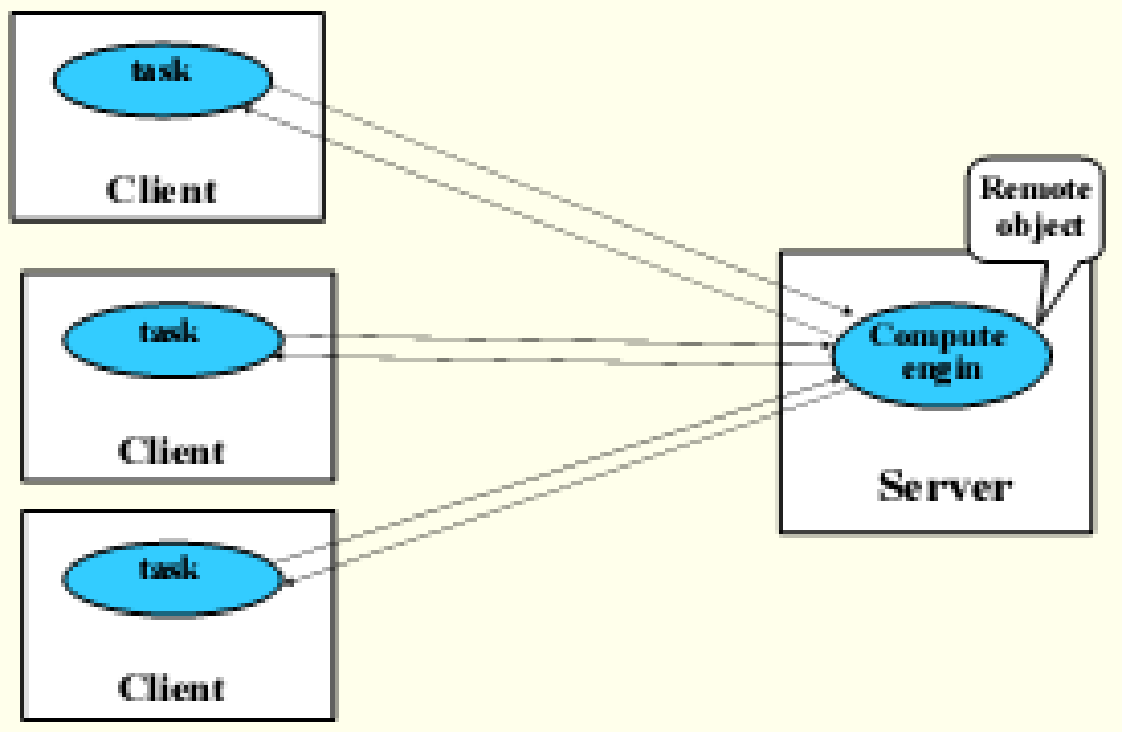


Java RMI实例研究---工作过程



一个RMI的分布式应用的实例

- 一个分布式计算引擎提供计算能力，客户端将计算任务提交给引擎计算，提交的任务要包括:计算步骤，计算引擎将计算结果返回。



一个RMI的分布式应用的实例

用RMI编写一个分布式应用，有以下三个核心问题：

- 定位远程对象

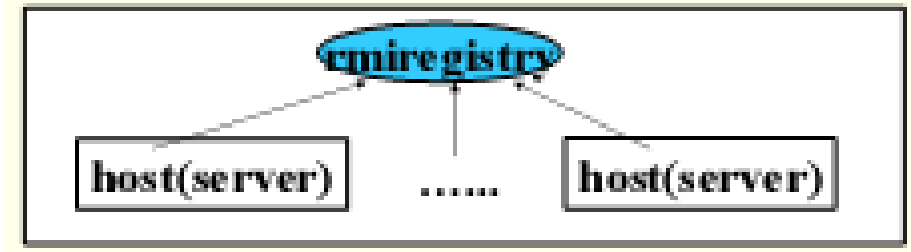
  1.一个应用可以利用RMI的名字服务功能注册其远程对象。

  2.可以象操作普通对象一样传送并返回一个运程对象的引用。

- 与远程对象通信：
  - 底层的通信由RMI实现，对于系统开发人员来说，远程调用和标准的Java方法调用没有什么区别。
- 为需要传递的对象装载类的字节码
  - RMI允许调用者向远程对象传递一个对象，因此RMI提供这种装载对象的机制。

一个RMI的分布式应用的实例

- 分布特性:
  - Engin独立开发，并运行，task后定义。写engin时不对执行什么任务作任何规定。任务可以是任意定制的
- 前提条件:
  - 定义任务的类，要规定任务的实现步骤，使得这个任务能够提交给engin去执行，使用server上的CPU资源
- 技术支持:RMI的动态装载功能.



一个RMI的分布式应用的实例

- 设计一个服务器
  - 核心协议:提交任务，执行任务，返回结果
  - 协议的表现形式:engin　task
  - 不同类型的任务，只要他们实现了Task类型，就可以在engin上运行.
  - 实现这个接口的类，可以包含任何任务计算需要的数据以及和任何任务计算需要的方法

一个RMI的分布式应用的实例

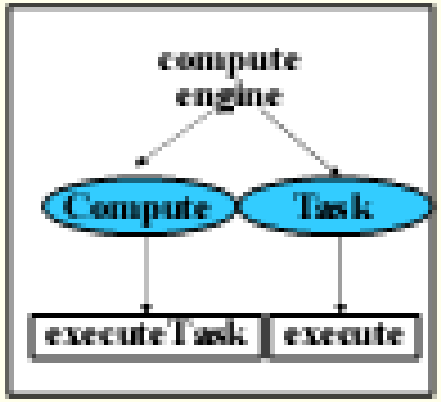
- Compute engine的设计要考虑以下问题:

1. Compute engine是一个类 ComputeEngine，它实现了Compute接口，只要调用该类的方法executeTask，任务就能提交上来.
2. 提交任务的Client端程序并不知道任务是被下载到engin上执行的.因此client在定义任务时并不需要包含如何安装的server端的代码.
3. 返回类型是对象，如果结果是基本类型，需要转化成相应的对等类.
4. 用规定任务如何执行的代码填写execute方法.

一个RMI的分布式应用的实例

❑定义远程接口

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Compute extends Remote
{   Object executeTask(Task t)
                    throws RemoteException;}
```




```
package compute;
import java.io.Serializable;
public interface Task extends
Serializable
{   Object execute();}
```

第二个接口:定义一个task类型,作为参数传给executeTask方法,

规定了engin与它的任务之间的接口,以及如何启动它的任务.它不是一个远程接口

一个RMI的分布式应用的实例

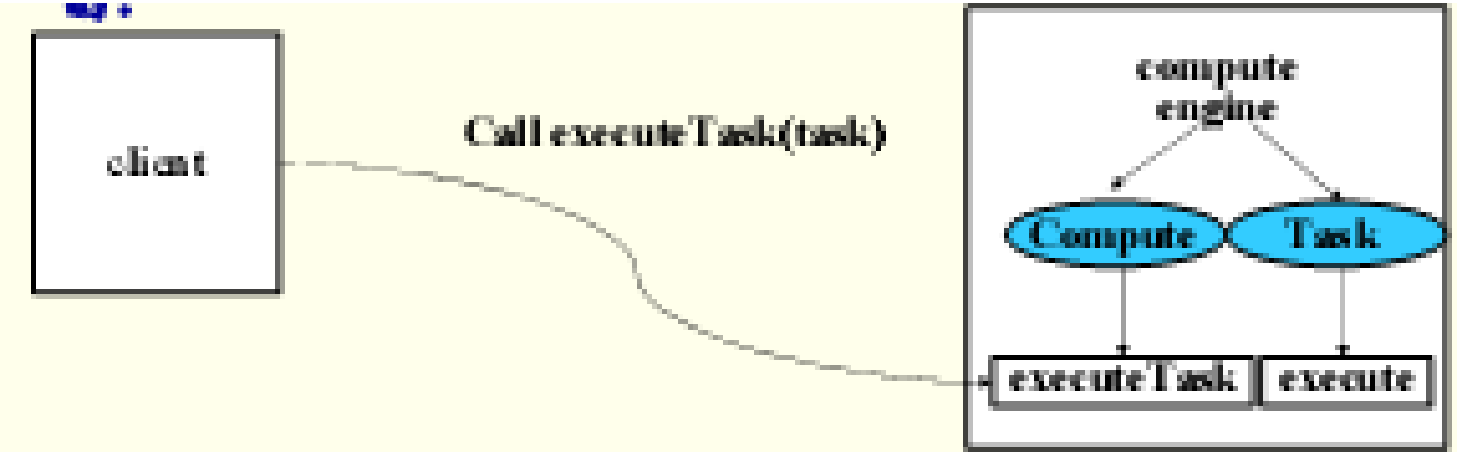
一般说来，实现一个远程接口的类至少有以下步骤:

1. 声明远程接口
2. 为远程对象定义构造函数
3. 实现远程方法
4. engin中端创建对象的工作可以在实现远程接口类的main函数中实现:
   - 创建并安装安全管理器
   - 创建一个或更多的远程对象的实例
   - 至少注册一个远程对象

```
package engine;
import  java.rmi.*; import java.rmi.server.*; import compute.*;
public class ComputeEngine extends UnicastRemoteObject
implements Compute
{ public ComputeEngine() throws RemoteException
{     super();    }
public Object executeTask(Task t)
{          return t.execute();  }
public static void main(String[] args)
{ if (System.getSecurityManager() == null)
{  System.setSecurityManager(new RMISecurityManager()); }
String name = "//host/Compute";
try { Compute engine = new ComputeEngine();
Naming.rebind(name，  engine);
System.out.println("ComputeEngine bound");
} catch (Exception e) { System.err.println("ComputeEngine
exception: " +  e.getMessage());    e.printStackTrace();  }}
```

一个RMI的分布式应用的实例

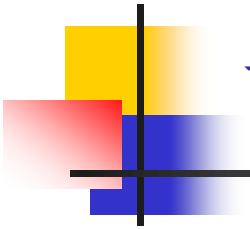
- 在构造函数中，通过super()， a UnicastRemoteObject被启动，即它可以侦听客户端来的请求输入
- 只有一个远程方法，参数是客户端远程调用这个方法时传来的任务.这个任务被下载到engin，远程方法的内容就是调用客户端任务的方法，并把结果回送给调用者.实际上这个结果是在客户的任务的方法中体现的.



一个RMI的分布式应用的实例

- 参数传递规则:

1. 远程对象通常通过引用传递.一个远程对象的引用是一个stub，它是客户端的代理. 它实现远程对象中的远程接口的内容
2. 本地对象通过串行化拷贝到目的. 如果不作制定，对象的所有成员都将被拷贝.



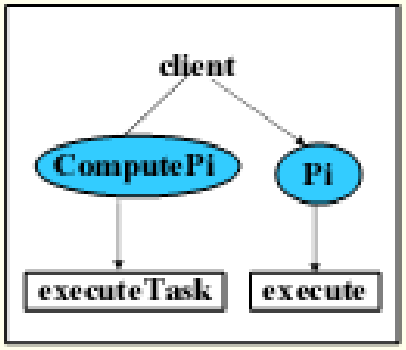
一个RMI的分布式应用的实例

一旦服务器用RMI注册了，main方法 就存在了，不需要一个守护线程工作维 护服务器的工作状态，只要有一个computer engin的引用在另一个虚拟机，computer engin就不会关闭

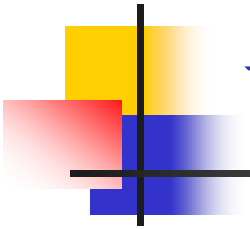
一个RMI的分布式应用的实例

- 实现一个客户程序
- 目标：创建一个任务，并规定如何执行这个任务。

```
package compute;
public interface Task extends
java.io.Serializable {
    Object execute();
}
```



- task不是远程接口，但是需要传递到服务器，因此用序列化。



一个RMI的分布式应用的实例

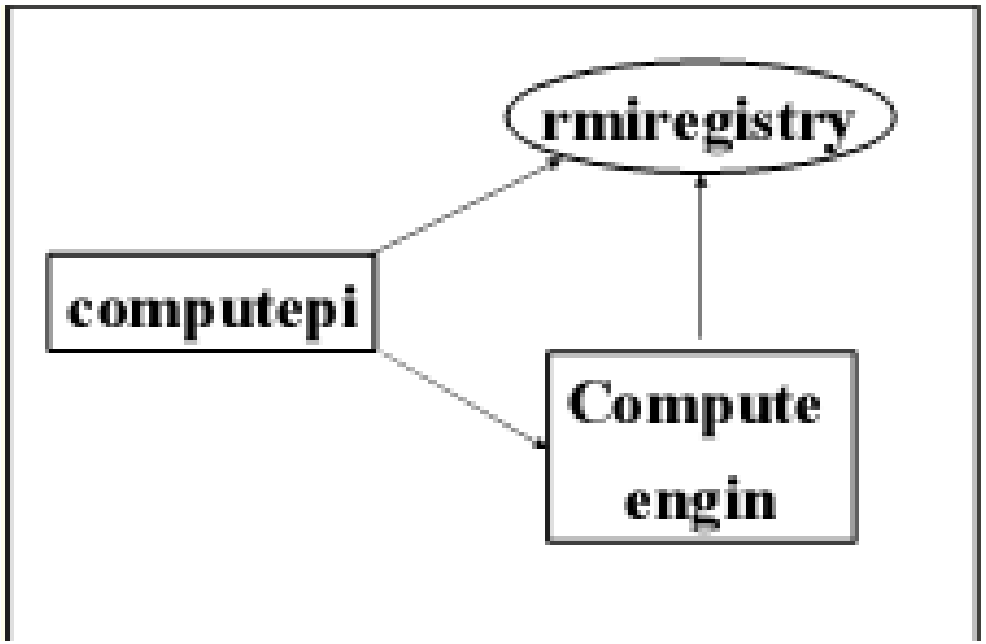
- computePi的作用

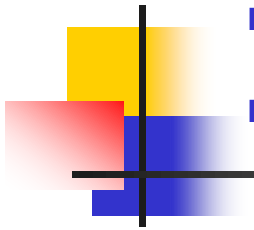
1. 调用远程方法，获得这个方法的引用
2. 生成一个任务
3. 要求任务被执行

```
package client;
mport java.rmi.*; import java.math.*; import compute.*;
public class ComputePi {
        public static void main(String args[]) {
        if (System.getSecurityManager() == null)
        {  System.setSecurityManager(new
        RMISecurityManager()); }
        try  { String name = "//" + args[0] + "/Compute";
        Compute comp = (Compute) Naming.lookup(name);
        Pi task = new Pi(Integer.parseInt(args[1]));
        BigDecimal pi = (BigDecimal) (comp.executeTask(task));
        System.out.println(pi);
        } catch (Exception e) {
        System.err.println("ComputePi exception: " +
        e.getMessage());
        e.printStackTrace();

    }
}
```

一个RMI的分布式应用的实例

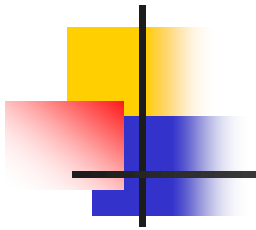




```
package client;
import compute.*; import java.math.*;
public class Pi implements Task {
  private static final BigDecimal ZERO =
  BigDecimal.valueOf(0);
  private static final BigDecimal ONE =
  BigDecimal.valueOf(1);
  private static final BigDecimal FOUR =
  BigDecimal.valueOf(4);
  private static final int roundingMode =
  BigDecimal.ROUND_HALF_EVEN;
public Pi(int digits)
{    this.digits = digits; }
```

```
public Object execute()
{       return computePi(digits);      }

*****************************************************
*
*       pi/4 = 4*arctan(1/5) - arctan(1/239)
*****************************************************
*
public static BigDecimal computePi(int digits) {
int scale = digits + 5;
BigDecimal arctan1_5 = arctan(5，  scale);
BigDecimal arctan1_239 = arctan(239，scale);
BigDecimal pi =
arctan1_5.multiply(FOUR).subtract(arctan1_239).multiply
(FOUR);
return pi.setScale(digits，  BigDecimal.ROUND_HALF_UP);   }
```

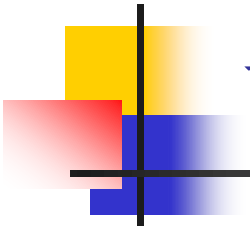


- /**
- * Compute the value， in radians， of the arctangent of
- * the inverse of the supplied integer to the speficied
- * number of digits after the decimal point. The value
- * is computed using the power series expansion for the
- * arctangent:
- * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 +
- * (x^9)/9 ...
- */

```
public static BigDecimal arctan(int inverseX， int scale)
{ BigDecimal result， numer， term;
BigDecimal invX = BigDecimal.valueOf(inverseX);
BigDecimal invX2 =
BigDecimal.valueOf(inverseX * inverseX);
numer = ONE.divide(invX， scale， roundingMode);
result = numer; int i = 1;
do { numer =numer.divide(invX2， scale， roundingMode);
int denom = 2 * i + 1;
term = numer.divide(BigDecimal.valueOf(denom)，
scale， roundingMode);
if ((i % 2) != 0) { result = result.subtract(term); }
else { result = result.add(term); }
i++;
} while (term.compareTo(ZERO) != 0);
return result;
```

一个RMI的分布式应用的实例

- engin一方并不需要事先知道它要计算的任务的任何特征。
- 由于Rmi的存在，系统可以做到：
  1. 可以直接通过名字定位远程方法的位置
  2. 参数的形式将一个对象传递给一个远程方法
  3. 可以使一个对象到另外一个虚拟机上运行
  4. 计算结果可以返回

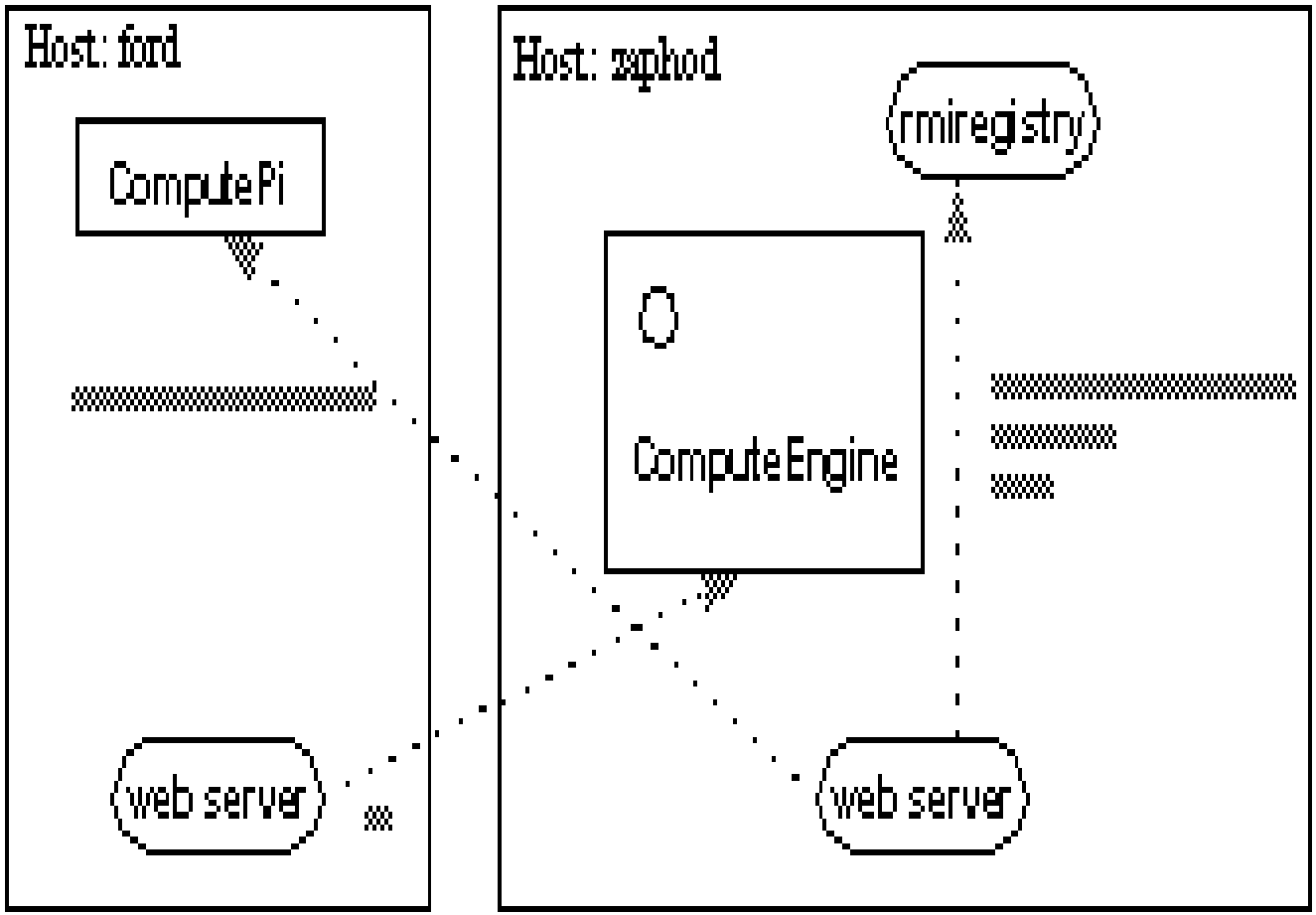


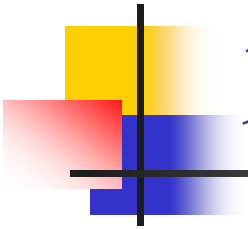
一个RMI的分布式应用的实例

将接口，远程对象，客户代码分成三个程序包：

1. compute (Compute and Task interfaces)
2. engine (ComputeEngine implementation class and its stub)
3. client (ComputePi client code and Pi task implementation)

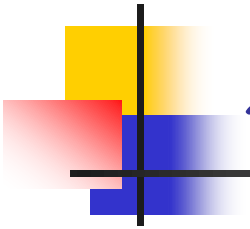
一个RMI的分布式应用的实例





第4章:分布式对象和远程调用

- 引言
- 分布式对象间的通信
- 远程过程调用Remote procedure call
- 事件与通知Events and notifications
- Java RMI 实例研究
- 总结



总结

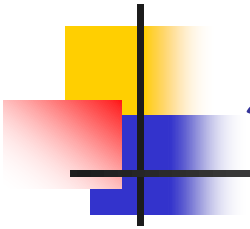
- 远程分布式对象的模型
  - 远程接口、远程对象引用，远程调用
- 利用分布式对象技术进行系统设计的基本方法
  - RMI
  - Sun RPC
- 基于事件通知机制的分布式系统的特点

思考题（练习）

- 使用例子
  - https://blog.csdn.net/qq_28081453/article/details/83279066?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.control&dist_request_id=&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.control
- 原理例子
  - https://blog.csdn.net/xinghun_4/article/details/45787549?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-9.control&dist_request_id=1328656.12727.16158993915174283&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-9.control

思考题（阅读）

- 比较SOA、微服务结构、RMI、RPC、Rest、RestFul、SOAP、WebService等概念的用途和优缺点
  - https://blog.csdn.net/u011474078/article/details/81427579?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-1.control&dist_request_id=&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-1.control



思考题

- C 语言中，.h文件的作用是什么？有编译的什么阶段会使用它们？尝试跟IDL做比较。
- 静态连接，动态链接分别主要解决什么问题，类比如果动态加载一个对象，需要解决哪些问题.
- 回调函数，与反射有类似性吗？
- 依赖注入，控制反转能跟以上两个问题类比吗？