KBENGINE
A MMOG ENGINE OF SERVER

KBEngine 技术概览

开源游戏服务端引擎

http://kbengine.org

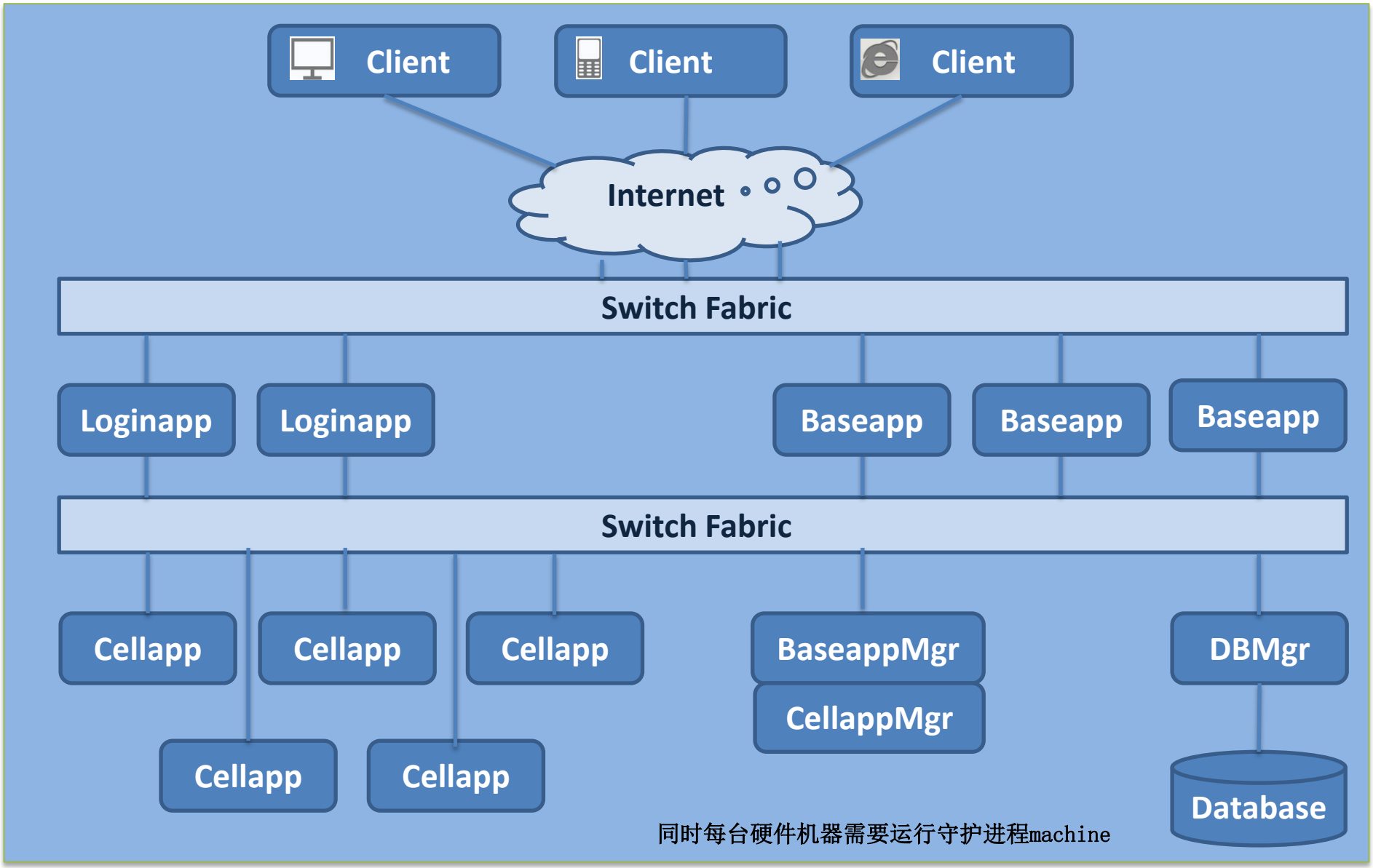
概要

- KBEngine 服务器概览
- 实现一个Entity
- Entity的通信
- Entity核心部分
- Cell 功能集
- 服务器设置和维护
- 服务器调试
- 服务器的profiling和压力测试

第一章

KBEngine 服务器概览

KBEngine 服务器架构



Loginapp进程

- 与客户端的第一个连接点
- 固定的端口
- 初始通信时加密
  - 公用密钥对(任意长度的密钥)
  - 用户名 / 密码
- 使用多个Loginapps使得负载均衡
  - DNS轮流调度

Baseapp进程

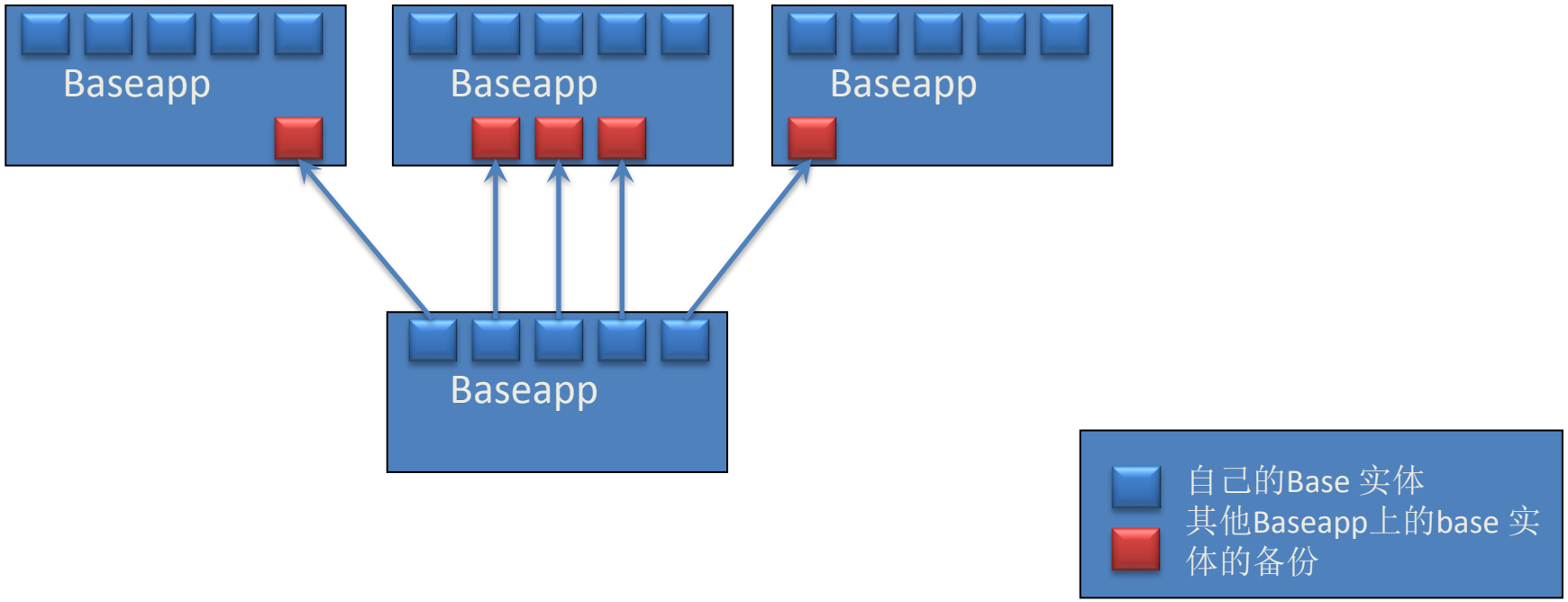
- 与客户端通信的固定点
- 客户端与Cellapp通信的中介
- 与客户端的连接均衡地分担在各Baseapp间
- 用于处理没有空间位置属性的Entity
  - 拍卖行
  - 公会管理
  - 管理器
- 每个Baseapp同时担任着为其它Baseapp容错的角色
- 通常一个CPU / 核 上处理一个Baseapp

Base Entity(实体)

- Baseapp上有两种实体
  - Base
  - Proxy
- Base
  - 通常的游戏Entity
  - 例如: 存储在数据库里的NPC, 拍卖行...
- Proxy
  - 与客户端连接
  - C++继承自KBEngine.Base
  - 特殊的Base

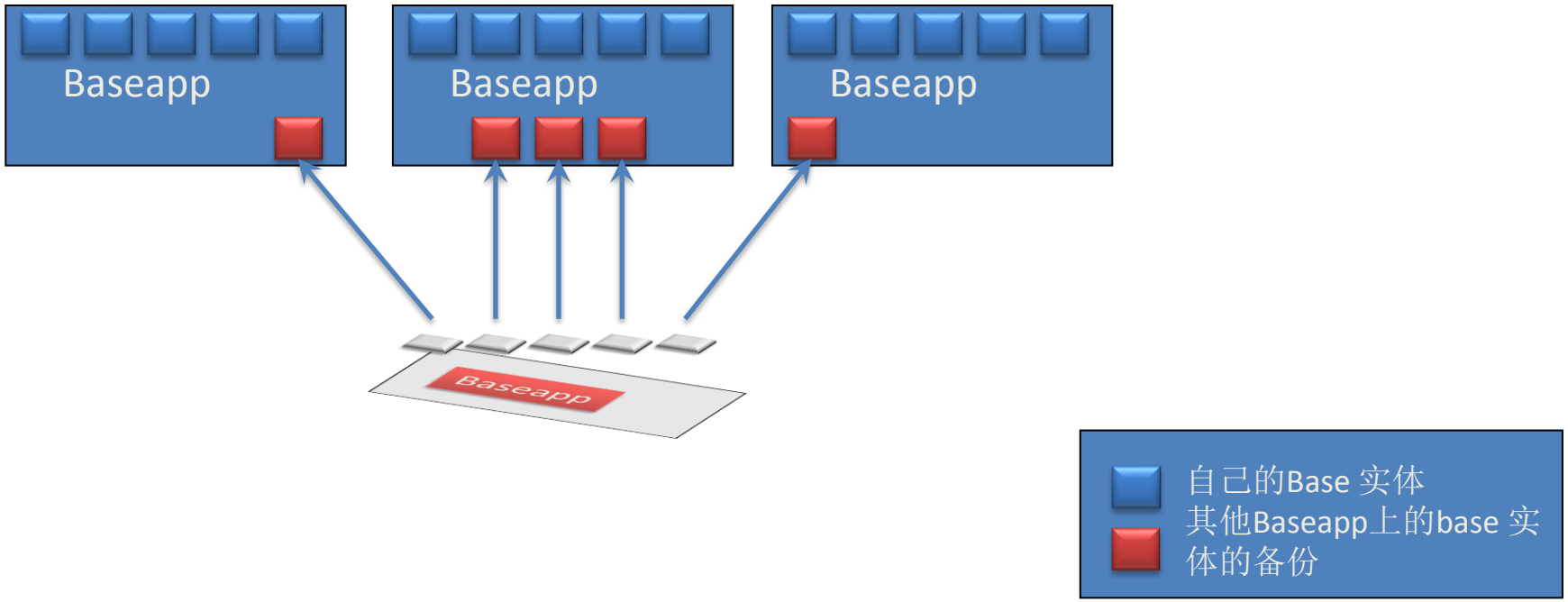
Baseapp 容错处理

- 备份entity到其它的Baseapps



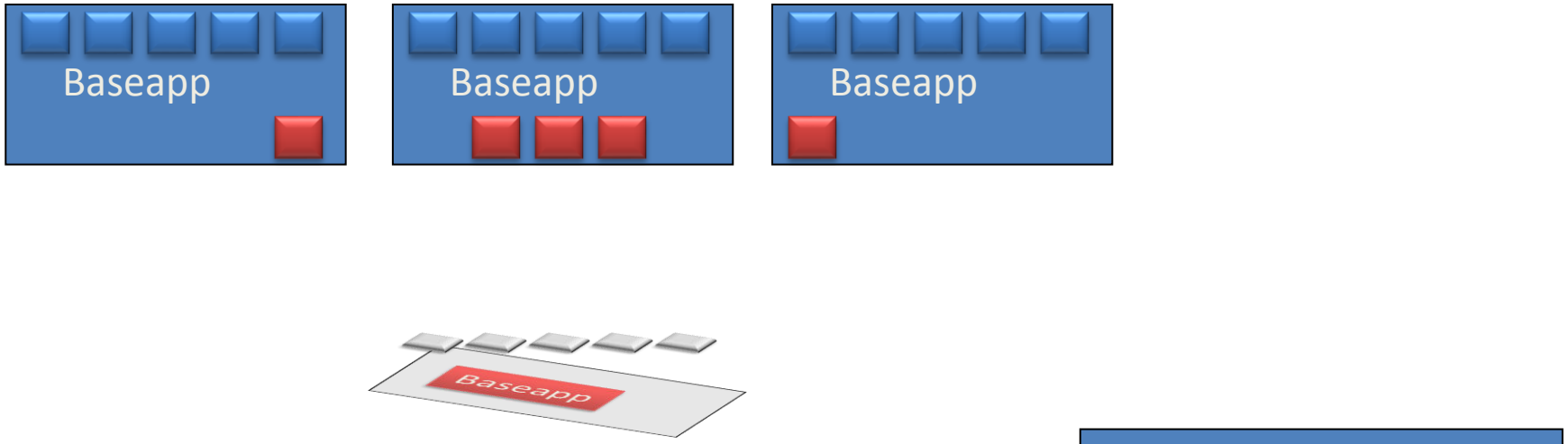
Baseapp 容错处理

- Baseapp crash后变得不可用



Baseapp 容错处理

- 灾难发生后快速切换到其他备份的Baseapp



Baseapp 容错处理

- 与Crash的Baseapp连接的客户端会被断开连接

  所有的数据都被存储了

  当重新连接后，它们将继续与其原来的Entity 连接　(如果没有timeout的话)

Baseapp 的管理器(BaseappMgr)

- 负责管理Baseapp间的负载平衡
- 监视所有的Baseapp以实现各个Baseapp之间的容错
- 主要用于玩家登录分配和创建Entity
- 一个服务器群组有一个BaseappMgr实例

Cellapp进程

- 空间与位置数据的处理

  处理玩家交互的Space (空间、房间、场景...)

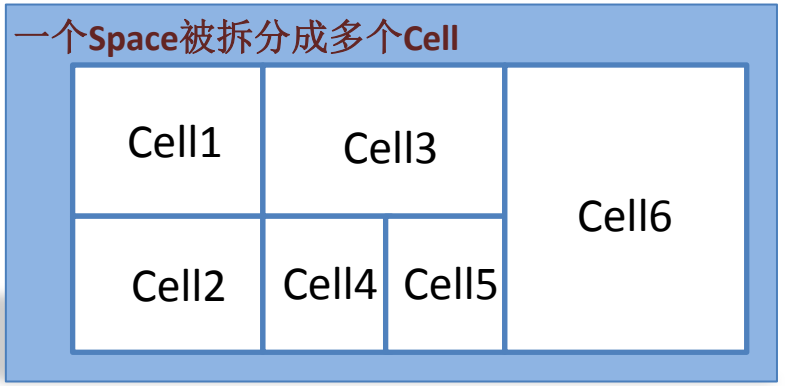
- 处理在Space内的Entity
- 处理Space内的一个区域 (Cell)

  一个Cellapp在一个Space上的Cell只会有一个(通常 进程占用一个CPU/核，多个Cell并没有意义)

- 一个Celapp有可能处理多个Space
- 通常一个CPU/核上处理一个Cellapp

Cells & Spaces

- 目前不支持Space拆分成多个Cell由Cellapps共同负载，因此本页可忽略
- Spaces通过Cells来实现平衡负载
- 每个Space至少含有一个Cell
- 每个Cell处理Space的一个区域
- Cell的边界根据Cell的负载而移动
- Cells不影响客户端的游戏体验



Cellapp主要负载的地方

- 管理的Entity的总数量
- Entity的通信的频率
  - 用户所调用的方法
  - 系统自动更新的属性
  - Entity的密集度
- Entity 脚本
- Entity的数据大小

Entity与Cell

- 每个space至少有一个Entity

  通常第一个Entity是SpaceEntity，用于让用户操控Space

- CellApp上的每个玩家Entity都有一个Witness对象

  Witness监视周围的Entity，将发生的事件消息同步到客户端

- Entity的兴趣范围(AOI)缺省是500M

  是可以自定义的，依赖于很多因素

Entity与Cell (本页跨Cell内容未实现)

- Entity穿越Cell边界是无缝的

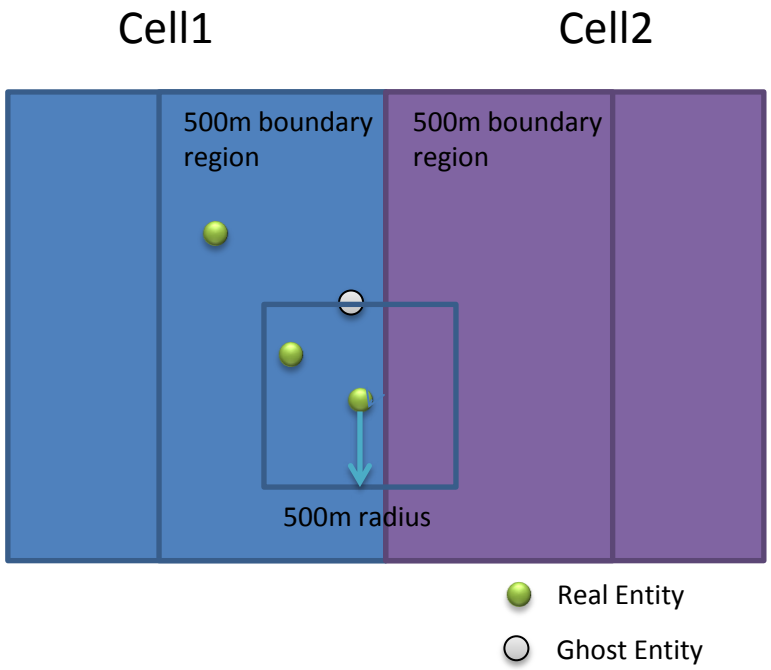
  客户端不会感觉到(穿越边界的发生)

- 每个Cell维护着一个list，存放着在其边界外沿的Entity

  Ghost entities

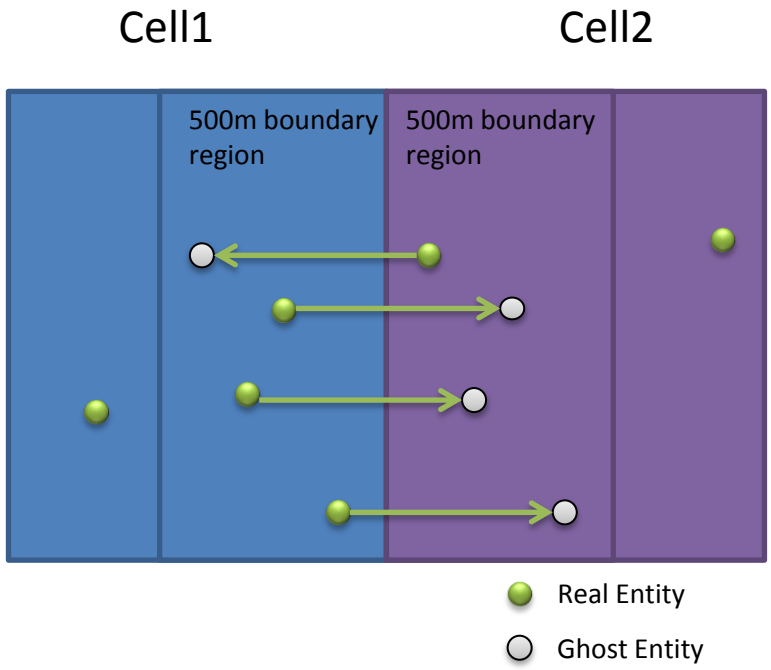
  半径500m，可配置

  大于等于AOI



Entity: Real与Ghost

- Real Entity是权威的Entity
- 一个Ghost Entity是从邻近的Cell的对应的Entity的部分数据的拷贝



Ghost Entity

- 解决跨越Cell边界的Entity的交互问题
- 方法调用

  转发给其Real Entity
- 属性

  一个属性可以是real only的, 例如: 将永远不会存在于ghost上

  如果一个属性对于客户端是可见的，那么该属性必须是可以ghost的,例如：当前的武器、等级、名称
- Ghost属性是只读的

  要更改属性值只能通过方法调用来更新其对应的Real Entity

Entity的数据更新

- 客户端实现LOD以加速渲染
- Cellapp实现LOD以减少:
  - 带宽的消耗
  - 每个Entity的CPU消耗
- LOD在Cellapp上的作用类似于在客户端的作用
  - 细节程度是相对于玩家entity与之的距离的
- 客户端Entity方法可以实现LOD
- Entity属性实现LOD可以避免不必要的通信到客户端
  - 当前的血量 (对于很远的距离(的Entity)来说是不可见的)

Cellapp管理器(CellappMgr)

- CellappMgr知道:
  - 所有的Cellapp (及它们的负载)
  - 所有的Cell边界
  - 所有Space
- 管理Cellapp的负载平衡
  - 告诉Cellapp们它们的Cell边界应该在哪里
- 把新建的Entity加入到正确的Cell上
- 一个服务器群组一个CellappMgr实例

数据库管理器(DBMgr)

- 管理Entity数据的数据库存储
- 负责数据库与其余的服务器间的Entity信息的通信
- 支持的数据库类型:
  - MySQL
  - MongoDB
  - Redis
  - … 你自己定制
- 最好独立的机器运行

Entity备份

- 存档
  - 在Baseapp间轮流调度处理
  - Baseapp向Cellapp要Entity的Cell部分的数据再定时转给DBMgr存储

KBEngine的机器Daemon(machine)

- Daemon用于监视服务器进程
- 每个服务器机器上有一个machine
- 启动/停止服务器进程
- 通知服务器群组各个进程的存活状态
- 监视机器的使用状态

  CPU / 内存 / 带宽

KBEngine服务端通常的操作

- 一个Baseapp，2个及以上Cellapp

  不同游戏不同情况

  早Profile，经常Profile

- 情况允许，应放在独立的机器的进程:

  DBMgr

  一些工具类进程

登录过程

- 客户端发登录请求
  - 指定IP/端口
- Loginapp收到登录请求
  - 解密请求消息(一些客户端也会选择不加密通讯，那么服务端不进行解密)
- Loginapp转发登录消息到DBMgr
- DBMgr验证用户名/密码
  - 查询数据库
- 转发请求到BaseappMgr
- BaseappMgr发送创建Player Entity的消息到负载最小的Baseapp
- Baseapp创建一个新的Proxy
  - 可能会创建一个新的Cell Entity
- Proxy的TCP端口被返回给客户端
  - 途径BaseappMgr, DBMgr, Loginapp

第二章

实现一个Entity

游戏项目资产库

- KBEngine引擎默认资产库

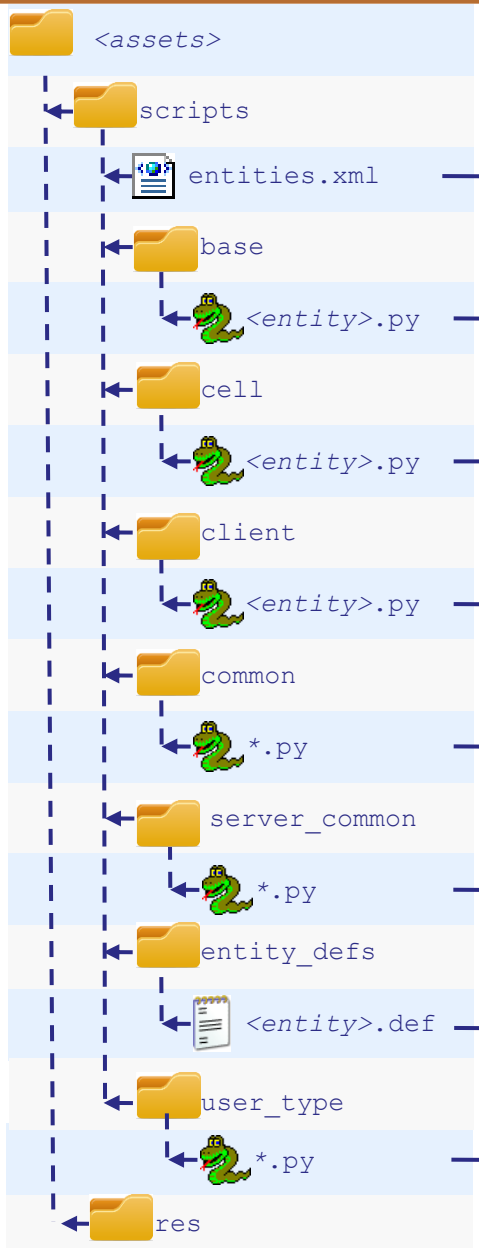
如果用户没有设置环境变量指向，引擎默认会尝试读取引擎根目录assets作为默认的资产库

资产库的概念类似于Unity3D中的Assets，不过其中一些文件夹名称结构被固定了

- 不同的项目是不同的资产库

要想引擎启动时读取到对应的项目资产库，必须在环境变量中制定

资产库文件夹结构



entities.xml → 列出了所有的entity

<entity>.py → Baseapp脚本

<entity>.py → Cellapp脚本

<entity>.py → Client脚本(只在包含Python解析器的原生环境有效，Unity3D等插件环境不需要在此实现)

*.py → 公用的script --> Cell, Base, Client 相互之间共用的实现函数

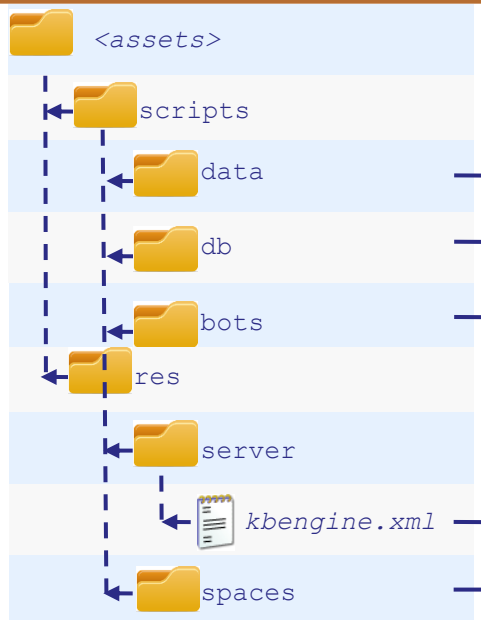
*.py → 公用的script --> Cell, Base 相互之间共用的实现函数

<entity>.def → 定义文件 ---> 定义Entity的属性和方法(XML)

*.py → 自定义类型脚本

实现属性和方法 (Python)

资产库文件夹结构



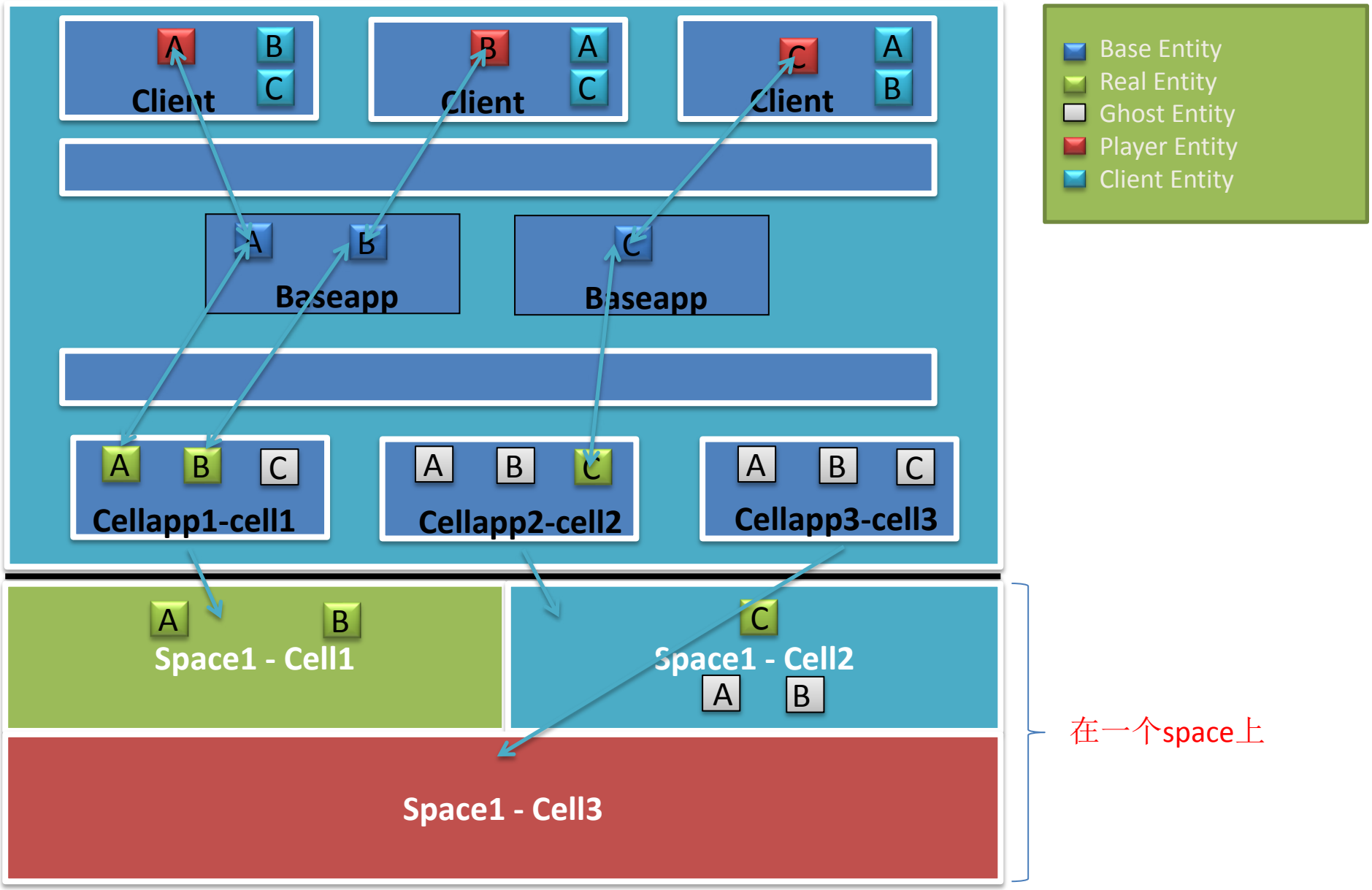
- data → 服务端逻辑数据文件，例如：策划导表数据
- db → 数据库扩展
- bots → 机器人压力测试，虚拟客户端脚本, 可以简化scripts/client下的脚本实现
- kbengine.xml → 服务端配置文件
- spaces → 空间的资源数据，例如：提供实体服务端导航的碰撞信息

Entity的实现

- 每个Entity必须:
  - 在entities.xml文件的列表里
  - 必须有一个<Entity_name>.def文件
  - 必须有<Entity_name>.py文件
- 每个Entity可以:
  - 有最多3个部分的实现 (Client/Cell/Base)
  - 使用common路径下的共享的脚本
- Client / Server的定义文件必须匹配

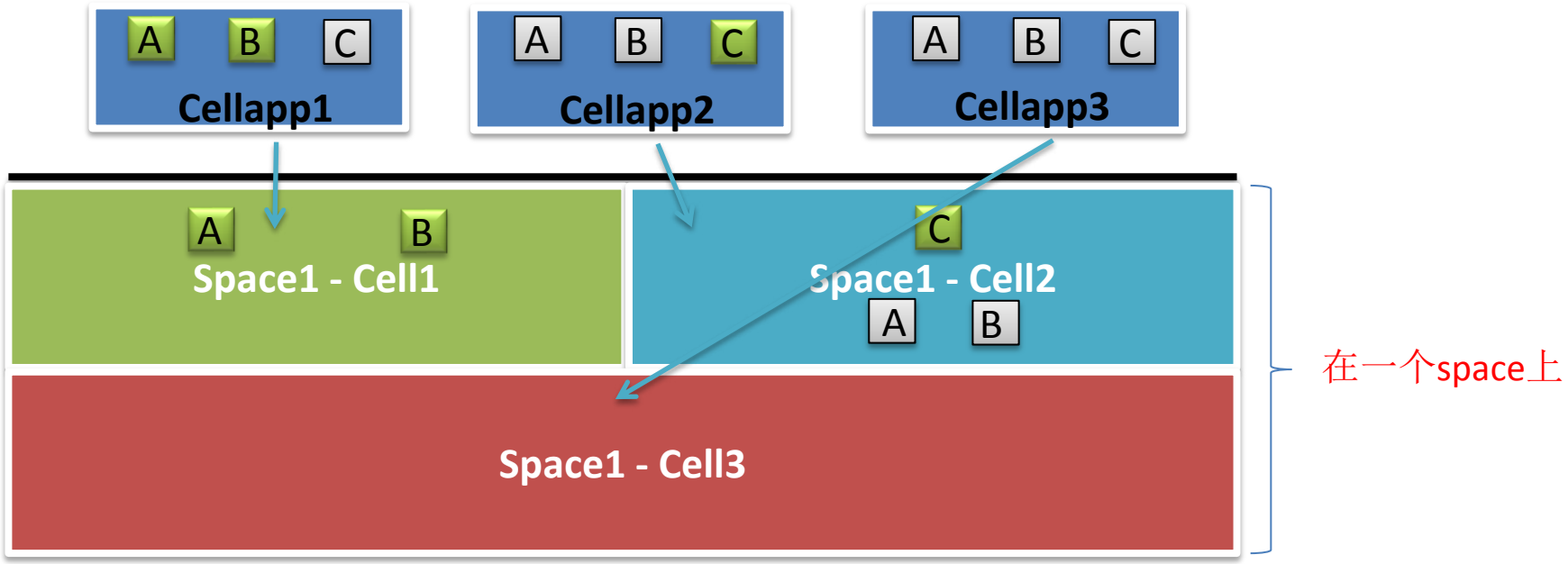
在一下插件环境，插件会根据协议MD5保证协议是最新的，当协议不匹配时会从服务端网络导入并存储到本地

分布式的Entity


Client
Client
Client
Baseapp
Baseapp
Cellapp1-cell1
Cellapp2-cell2
Cellapp3-cell3
Space1 - Cell1
Space1 - Cell2
Space1 - Cell3
Base Entity
Real Entity
Ghost Entity
Player Entity
Client Entity
在一个space上

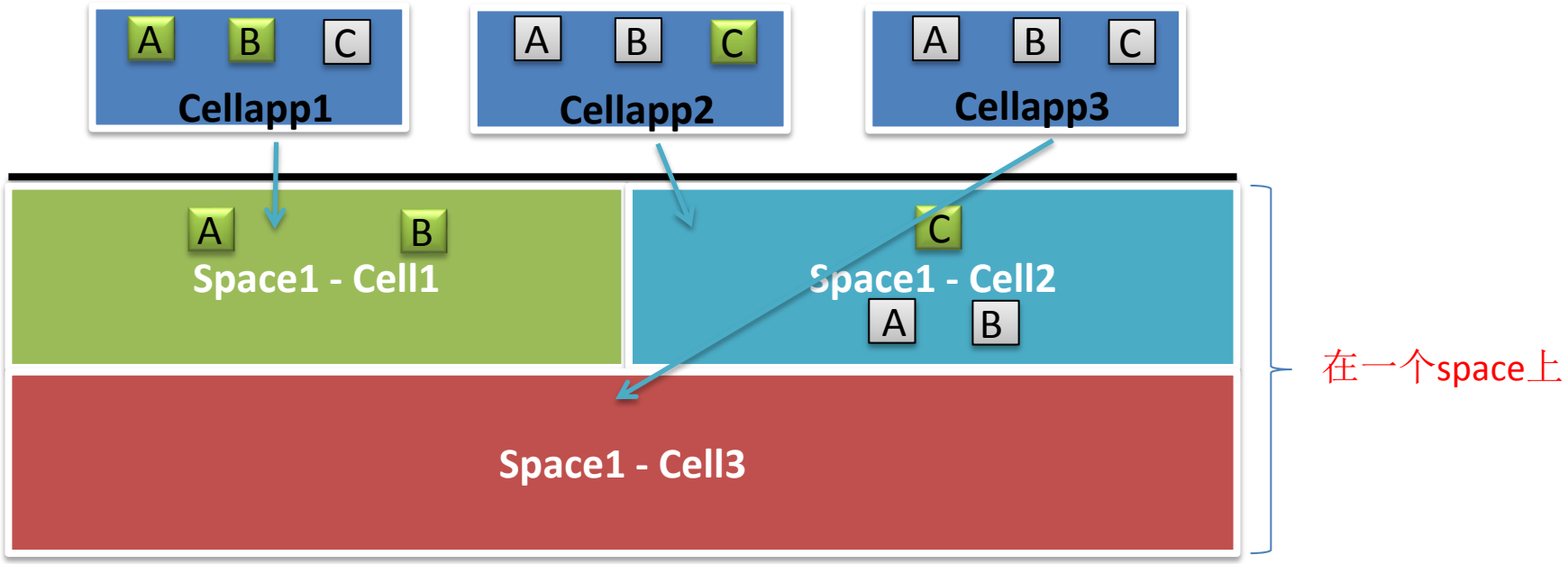
分布式的Entity

- CellApp1,CellApp2和CellApp3都各自有一个Space1的Cell
- 三个Entity A,B和C都在Space1



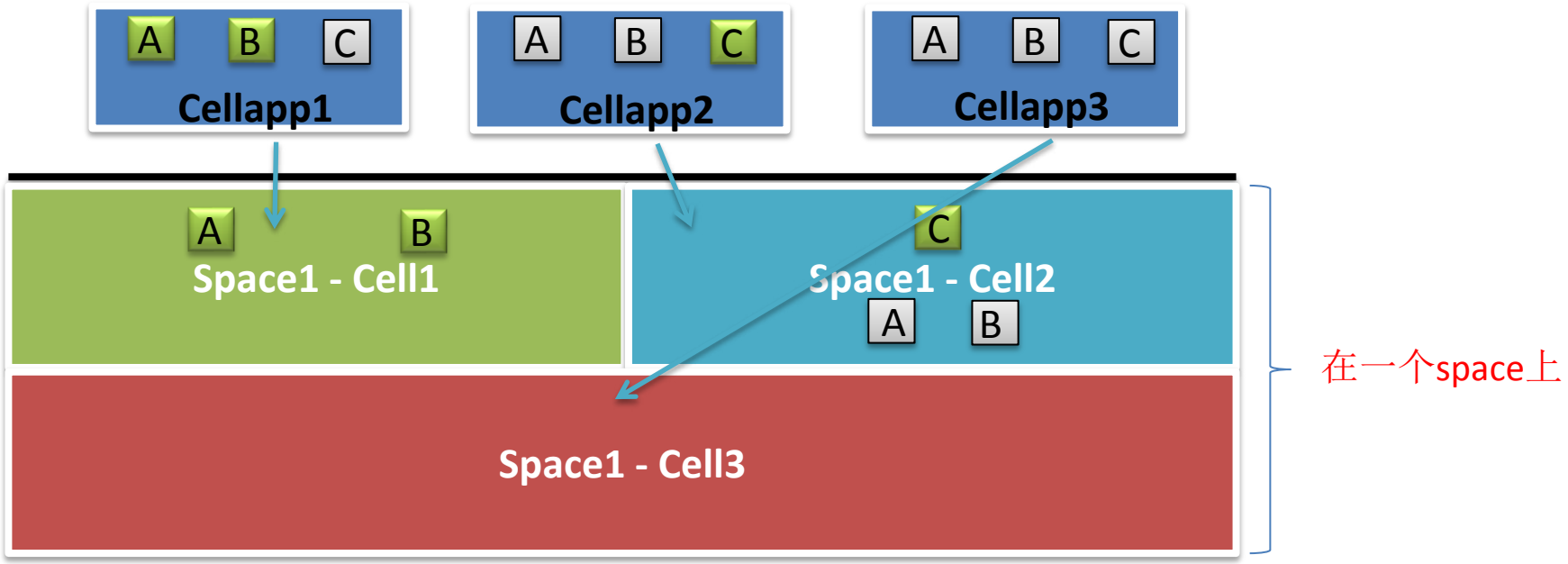
分布式的Entity-从Cellapp1来看

- Space1的CellApp1的Cell：
  - A和B是Real Entity
  - C是一个从CellApp2上ghost来的ghost Entity



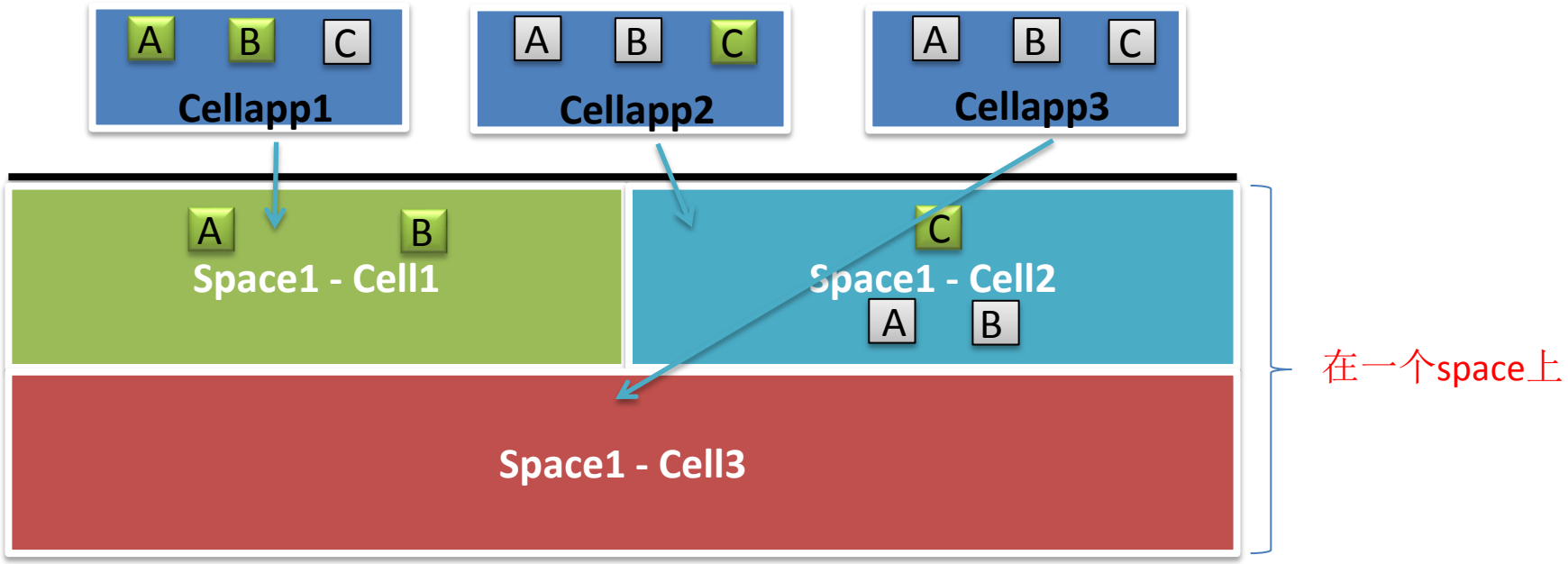
分布式的Entity-从Cellapp2来看

- Space1的CellApp2的Cell：
  - C是Real Entity
  - A和B是一个从CellApp1上ghost来的ghost Entity



分布式的Entity-从Cellapp3来看

- Space1的CellApp3的Cell：
  - A和B是一个从CellApp1上ghost来的ghost Entity
  - C是一个从CellApp2上ghost来的ghost Entity



简单的Entity

```
Account.def:
------------

    <root>
                <Properties>
                </Properties>

                <ClientMethods>
                </ClientMethods>

                <BaseMethods>
                </BaseMethods>

                <CellMethods>
                </CellMethods>
    </root>
```

Entity的继承

- Entity定义文件支持继承

  `<assets>/scripts/entity_defs/interfaces`

- 两种继承机制:

  `<Parent>`

  继承所有的东西

  属性 / 方法

  Volatile 属性定义

  LOD 级别

  简单级别的继承

  `<Implements>`

  继承属性和方法

  多级别的继承

Avatar的定义

```
<root>
    <Volatile>
        <position/>
        <!--<position> 0 </position> Don't update-->
        <yaw/>
        <!--<pitch> 20 </pitch>-->
        <pitch/>
        <roll/>
    </Volatile>

    <Implements>
        <Interface>     GameObject              </Interface>
        <Interface>     State                   </Interface>
    </Implements>

    <Properties>
        <roleType>
            <Type>          UINT8           </Type>
            <Flags>         BASE            </Flags>
            <Persistent>    true            </Persistent>
        </roleType>
    </Properties>

    <BaseMethods>
        <createCell>
            <Arg>           MAILBOX         </Arg>
        </createCell>
    </BaseMethods>

    <CellMethods>
        <jump>
            <Exposed/>
        </jump>
    </CellMethods>

    <ClientMethods>
        <onJump/>
    </ClientMethods>
</root>
```

Entity的属性

- 类型
  - 像所有语言一样
  - 为网络传输/数据库存储标准化
- 缺省值
  - 由类型决定
  - 多可以在定义文件里覆盖
- 广播形式的标志
- Detail Level
- Volatile 信息
- 是否存储到数据库

Entity属性

- Cell上的属性
  - Entity数据被频繁访问
  - 当跨越Cell Boundary时数据会被复制（到新的Cell）
  - 数据备份到Base
  - 数据改变时通知客户端:
    - 属性的改变
    - 当一个entity进入玩家的AOI时
- Base上的属性
  - 更复杂/访问较少
  - 数据改变时通知客户端

Entity属性

- Client上的属性
  - 可访问部分的server属性
  - 属性值从Cell上发布得来
  - Cell属性改变会触发set_<property>()
  - 例如:

```
def set_HP( self, old ):
  if self.HP == 0 and old > 0:
    self.doDeath()
```

Entity定义数据类型

- 简单类型
  - INT8 / UINT8
  - FLOAT32 / FLOAT64
  - STRING
  - VECTOR3
  - …
- 序列类型
  - ARRAY
  - TUPLE

Entity定义数据类型

```
<root>
   <Properties>
      <name>
         <Type>              STRING          </Type>
      </name>

      <items>
         <Type>              ARRAY
               <of>          UINT8            </of>
          </Type>
      </items>
   </Properties>
   ...
</root>
```

Entity定义数据类型

- 复杂类型
  - FIXED_DICT
    - Dictionary型的对象
    - 固定的key集
  - PYTHON
    - 比FIXED_DICT低效
    - 可以支持任何Python数据类型
    - 安全性
      (读取客户端传来的数据来序列化Python对象)
    - 使用Python的pickle模块
    - Unity3D等插件环境不应该将该属性类型传输到客户端
      (C#无法解析)

Entity定义数据类型

```
<root>

   <Properties>

      <CharacterInfos>
         <Type>  FIXED_DICT
            <Properties>
               <name>
                  <Type> STRING </Type>
               </name>

               <level>
                  <Type> UINT8  </Type>
               </ level >
            </Properties>
         </Type>
      </CharacterInfos>

    </Properties>

   ...
</root>
```

类型别名

- 可重用的类型自定义
  - `<assets>/scripts/entity_defs/alias.xml`

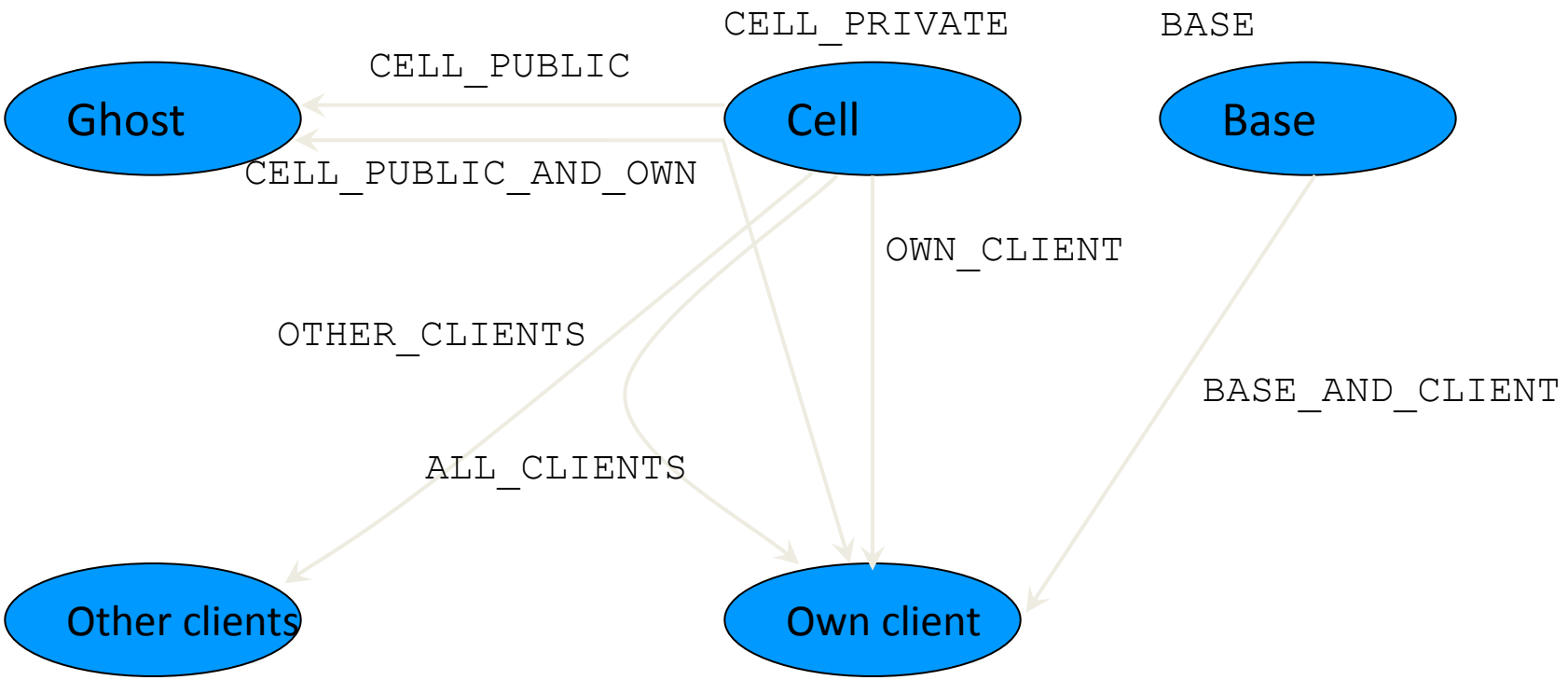
```
<SKILLID>                   INT32               </SKILLID>
<QUESTID>                   INT32               </QUESTID>
```

```
<EXAMPLES>          FIXED_DICT
            <Properties>
                        <k1>
                            <Persistent>    false       </Persistent>
                            <Type>          INT64       </Type>
                        </k1>

                        <k2>
                            <Type>          INT64       </Type>
                        </k2>
            </Properties>
</EXAMPLES>
```

Entity属性的发布

```
<root>
   <Properties>

      <name>
         <Type>    STRING          </Type>
         <Flags>   ??              </Flags>

      </name>

   </Properties>

   ...
</root>
```

Entity属性的发布



Entity属性的发布 - BASE

- 属于Base
- 只有Base可以访问

  Baseapp2和Baseapp3无法访问到Baseapp1中红色实体的BASE属性
- BASE属性的修改不会被广播
- 把它们定义在.def文件里就意味着它们会被定期的备份到其它Baseapp上和数据库里
- 例如:

  当前账号Entity记录玩家上次进入游戏所选择的角色DBID

  公会管理器记录的公会成员信息列表

Entity属性的发布 - BASE_AND_CLIENT

- 属于Base
- Base和自己的客户端可以访问

  Baseapp2和Baseapp3无法访问到Baseapp1中红色实体的BASE_AND_CLIENT属性
- 该类属性的值的改变也会被发布到其对应的自己的客户端的entity上。并且会有脚本的回调(`set_<property_name>()`)函数会被调用
- 例如:

  当前账号Entity记录玩家上次进入游戏所选择的角色DBID，但客户端也需要对所选角色进行表现

  很少用到

Entity属性的发布 – CELL_PRIVATE

- 属于Real Entity，只有Real Entity能访问
- Cellapp2和Cellapp3无法访问到Cellapp1红色实体的CELL_PRIVATE属性
- 在.def文件里定义它们就意味着在Cell的Entity从一个Cell换到另一个Cell上时这类的属性会被随着移植到新的Cell上。另外，这类的属性会被定期的备份到Base Entity上
- 例如:

  NPC AI ‘想法’

  Player的关于游戏play的属性，但是其它player不应该看到

Entity属性的发布 – CELL_PUBLIC

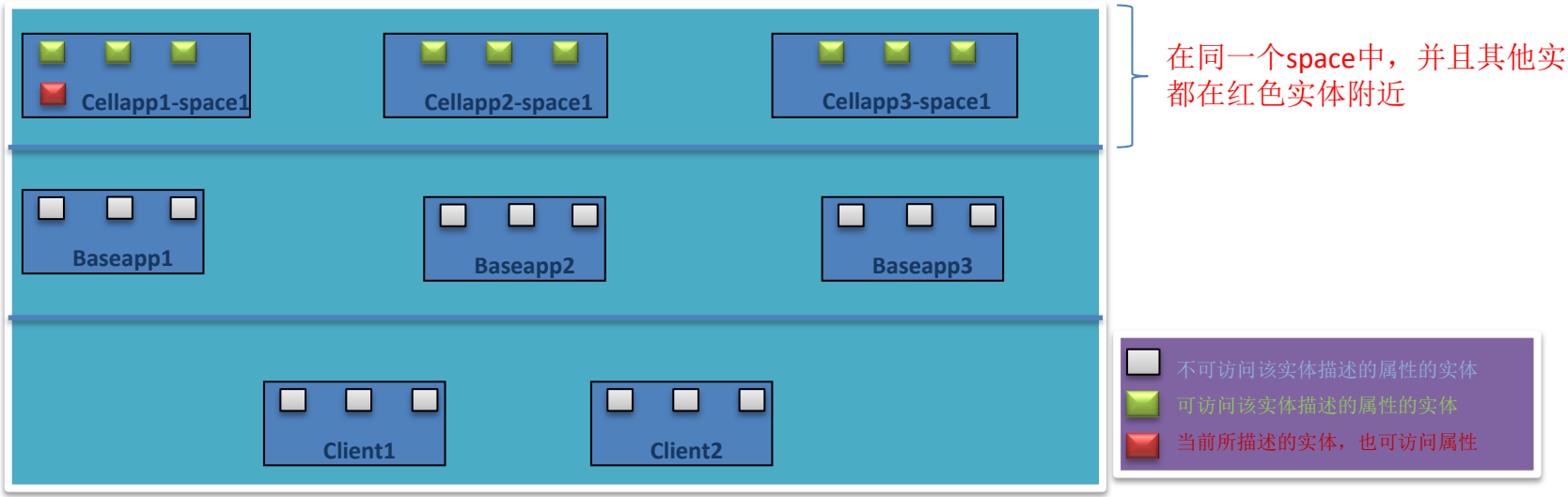
- 属于Real Entity

  它所属于的Real Entity和其对应的ghost Entity上都可以访问

- 该类属性的值的改变会被发布到其对应的ghost Entity上。在ghost Entity上这类属性只是只读的属性
- 例如:

  怪物的暴力级别

  NPC的等级



Entity属性的发布 – CELL_PUBLIC_AND_OWN

- 属于Real Entity

  它所属于的Real Entity和其对应的ghost Entity上都可以访问

- 该类属性的值的改变会被发布到其对应的ghost Entity上。在ghost Entity上这类属性只是只读的属性
- 该类属性的值的改变也会被发布到其对应的自己的客户端的entity上。并且会有脚本的回调(`set_<property_name>()`)函数会被调用
- 例如:

  Avatar的敌人列表，服务端其他实体AI可以检查Avatar敌人列表并协助战斗，客户端可以显示敌人列表中的仇恨值做排名，而其他客户端则不需要看到当前Avatar的仇恨列表

Entity属性的发布 – ALL_CLIENTS

- 属于Real Entity

  它所属于的Real Entity和其对应的ghost Entity上都可以访问
- 该类属性的值的改变会被发布到其对应的ghost Entity上。在ghost Entity上这类属性只是只读的属性
- 该类属性的值的改变也会被发布到其对应的自己的客户端的Entity上。并且会有脚本的回调(`set_<property_name>()`)函数会被调用
- 如果其它的玩家的AOI范围内有这个属性隶属的Entity，那么这个属性的值的改变也会被发布这些玩家的客户端的该Entity上。并且会有脚本的回调(`set_<property_name>()`)函数会被调用
- 例如:

  实体名称

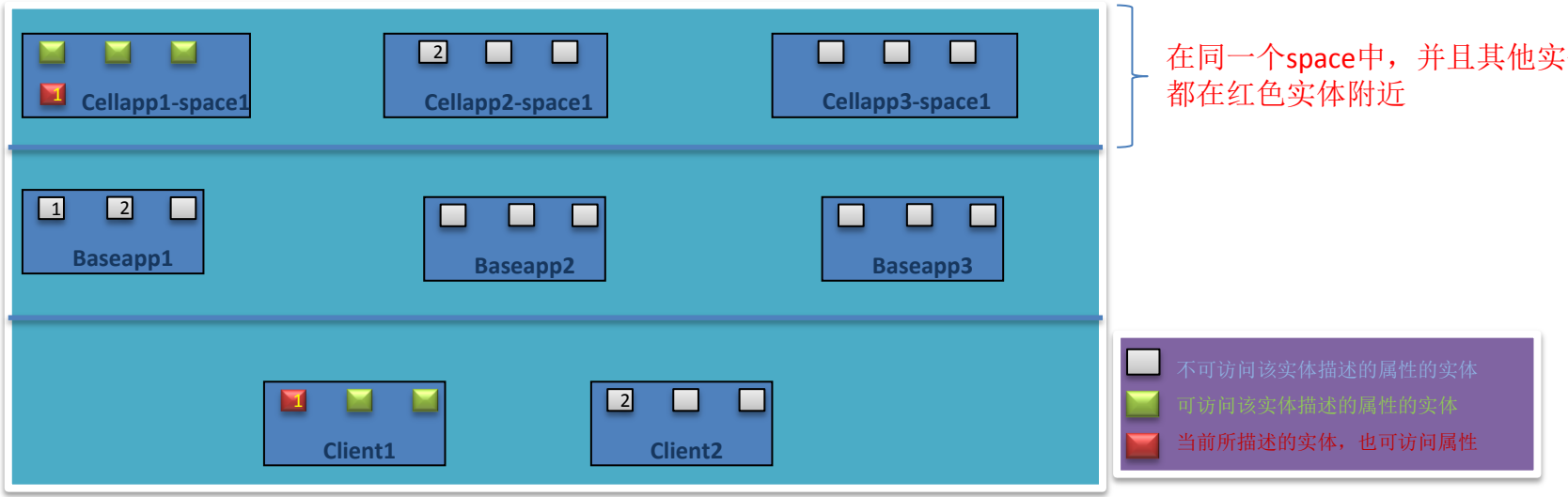
  实体血量与等级

Entity属性的发布 – OWN_CLIENT

- 属于Real Entity
- 它所属于的Real Entity和自己的客户端可以访问
- 该类属性的值的改变也会被发布到其对应的自己的客户端的Entity上。并且会有脚本的回调(`set_<property_name>()`)函数会被调用
- 例如:

  角色当前的敏捷、力量、智力属性，该属性用于计算角色最终的能力值，但其他实体不需要访问该属性，而自己的客户端需要在角色面板上显示这三个属性用于配点

  角色的经验值



Entity属性的发布 – OTHER_CLIENTS

- 属于Real Entity

  它所属于的Real Entity和其对应的ghost Entity上以及其他de 客户端都可以访问
- 该类属性的值的改变会被发布到其对应的ghost Entity上。在ghost Entity上这类属性只是只读的属性
- 如果其它的玩家的AOI范围内有这个属性隶属的Entity，那么这个属性的值的改变也会被发布这些玩家的客户端的该Entity上。并且会有脚本的回调(`set_<property_name>()`)函数会被调用
- 例如:
  - 动态的世界物品的状态 (如: 门, 按钮, 战利品)
  - 客户端本地已知某状态，只是想将状态广播给其他客户端

Volatile属性

- 优化的协议
- 仅仅对最近更新的数据值有兴趣
- Position (x,y,z)
- Yaw, Pitch, Roll

属性Detail Level(未实现)

- 影响属性更新到客户端
- 典型地用于可看见的属性
- 带宽节省机制
- 如果需要可以使用，但不是必须用
- 用<DetailLevel>指定
- Detail levels 又名 <LodLevels>

属性Detail Level(未实现)

```
<root>

   <LoDLevels>
      <level>  20  <label> NEAR    </label>  </level>
      <level> 100  <label> MEDIUM  </label>  </level>
      <level> 250  <label> FAR     </label>  </level>
   </LodLevels>

   <Properties>
      <name>
         <Type>          STRING       </Type>
         <Flags>         ALL_CLIENTS  </Flags>
         <DetailLevel>   NEAR   </DetailLevel>
      </name>
   </Properties>

   ...
</root>
```

Entity的数据保存

- 一些entity和它们的数据可能需要保存到数据库，这样就是服务器重启了这些数据也还有效
- 在属性上定义
- Entity被存到数据库里
- 自动在数据库里创建一个`self.databaseID`

```
<root>
   <Properties>
      <name>
         <Type>          STRING          </Type>
         <Flags>         ALL_CLIENTS     </Flags>
         <Persistent>    true            </Persistent>
      </name>
   </Properties>

   ...
</root>
```

Entity方法

- 分别定义在
  - Client / Cell / Base
- 必须定义参数
- Base / Cell方法可以被暴露给Client端使用
- Client方法可以指定一个最大的可调用范围
- 要远程的调用(跨域Client / Cell / Base)必须在定义文件(.def)里定义

Entity方法

```
<root>
   <Properties>
      ...
   </Properties>

   <ClientMethods>
     ...
   </ClientMethods>

   <BaseMethods>
      <addToFriendsList>
         <!-- Entity ID -->
         <Arg>  INT32  </Arg>

         <!-- Expose to client -->
         <Exposed />
      <addToFriendsList>
   </BaseMethods>

   <CellMethods>
      ...
   </CellMethods>
</root>
```

Entity暴露方法（允许Client调用）

- 不是所有的server方法都被暴露的
- 需要以<Exposed />声明
- 暴露的Cell方法
  - 自动的接收调用方的EntityID
  - 通常要检查是否
    `self.id == callerID`
- 暴露的Base方法
  - 只有自己的Client可以调用

Entity方法(本页未实现)

- Client方法LoD
  - 帮助减少客户端带宽的使用
  - 在近距离产生可视的效果
  - 当广播Client消息时很有用

```
<root>
   ...

   <ClientMethods>
     ...
     <smile>
       <DetailDistance> 30 </DetailDistance>
     </smile>
     ...
   </ClientMethods>

   ...
</root>
```

Entity方法

- Entity根据需要存在于Cell/Base/Client分布平台的一个或多个上。
- 如果在一个分布平台上没有Entity存在的需要，那么在这个平台上也不需要该Entity的Python脚本。

Entity分布式存在的例子

* 没有**Base**部分的**Entity**是不参与容错的

脚本开发的指导

- 尽可能的把负载放到BaseApp
- 尽可能减少需要保存到数据库的Entity的属性
- 避免过多调用`writeToDB()`
- 尽量减少复杂层级的数据
  - 如: 多维数组
- 如果脚本的执行时间超过1个game tick，会负面地影响服务器的效率

第三章

Entity的通信

Mailbox

- 指向远程进程的Entity
  - 如: 一个Entity的Cell部分
- 使得可以远程的调用函数（从一个进程调用另一个进程的函数）
  - 如: mb 是一个Cell entity mailbox

    ```
    mb.someMethod( a, b )
    ```

    会引发Real cell entity所在的进程的someMethod()的调用
- Intra-entity通信
  - 如: 从Cell部分到Base部分
- Inter-entity通信
  - 如: Entity A的Cell部分到Entity B的Base部分

Mailbox

- 不同的类型
  - Base
  - Cell
  - Client
  - 单步跳转
  - 多步跳转
    - 通过base到cell(xxx.base.cell.someMethod())
- 一些KBEngine方法可能只接收一些特定的类型的Mailbox
  - 细节请参考API文档

Mailbox

- Entity有mailbox成员变量
  - Client entity: `self.cell`, `self.base` (用于玩家)
  - Base entity: `self.cell`
  - Proxy entity: `self.cell`, `self.client`
  - Cell entity:
    - `self.base`
    - `self.ownClient`
    - `self.allClients`
    - `self.otherClients`

Mailbox

- 当一个Entity对象被传到一个有Mailbox参数的server方法时，Mailbox对象被自动地创建
- 例如:
  - Cell方法`talkToMe()`有一个`MAILBOX`参数
  - 在一个Cell上, EntityA调用:

```
entityB.talkToMe( self )
```

  - Entity A的mailbox被传到Entity B

```
def talkToMe( self, mailbox ):
    mailbox.sendMsg( “hello” )
```

  - Entity A的`sendMsg()`被调用（以“hello”为参数）

存储Mailbox

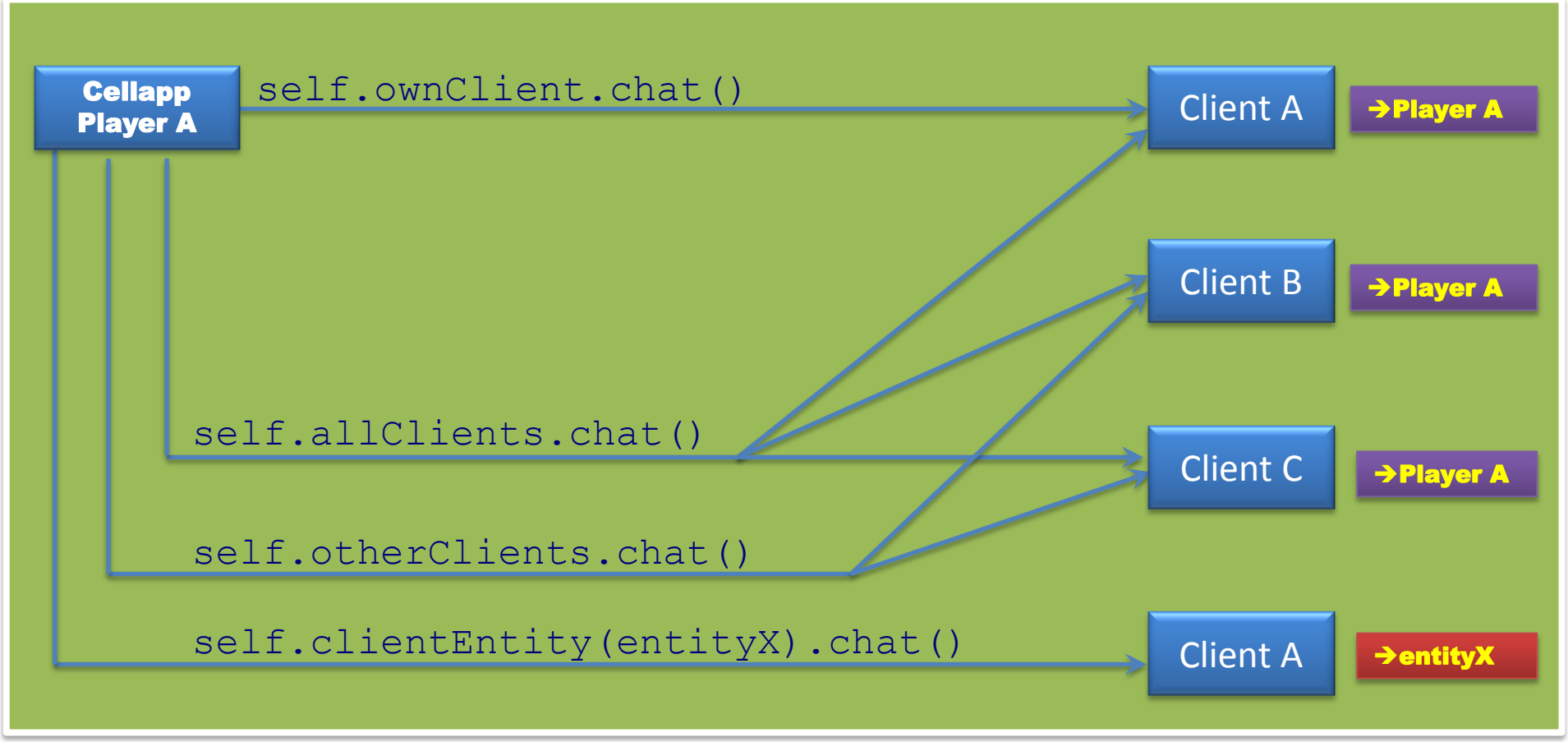
- Base的mailbox是在Entity的生命周期内都有效的
  - Base entity所在的Baseapp永远不改变
  - 可以用于长久的Entity间的通信
  - 如果储存一个mailbox，必须实现一个消息通知的机制（Entity删除时通知mailbox无效）
- Cell mailbox只在很短的时间内有效
  - Cell entity所在的Cellapp随时可能改变
  - 不要保存Cell **`MailBox`** 作为属性
  - 立即使用，立即释放

存储Mailbox

- 不能从Client传递Mailbox也不能传递Mailbox到Client
  - 不能信任Client
  - 取而代之用Entity ID
- Mailbox不能被存储到数据库里
  - 当Server重启后IP地址会被改变

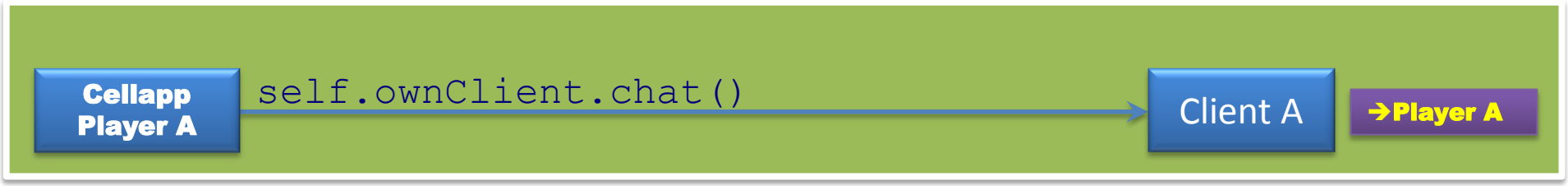
Cell到Client的通信

- Self是Player A
- Player在Baseapp上必须是一个 Proxy
- 这些MailBox不能被传递
- 消息由Baseapp中转到Client

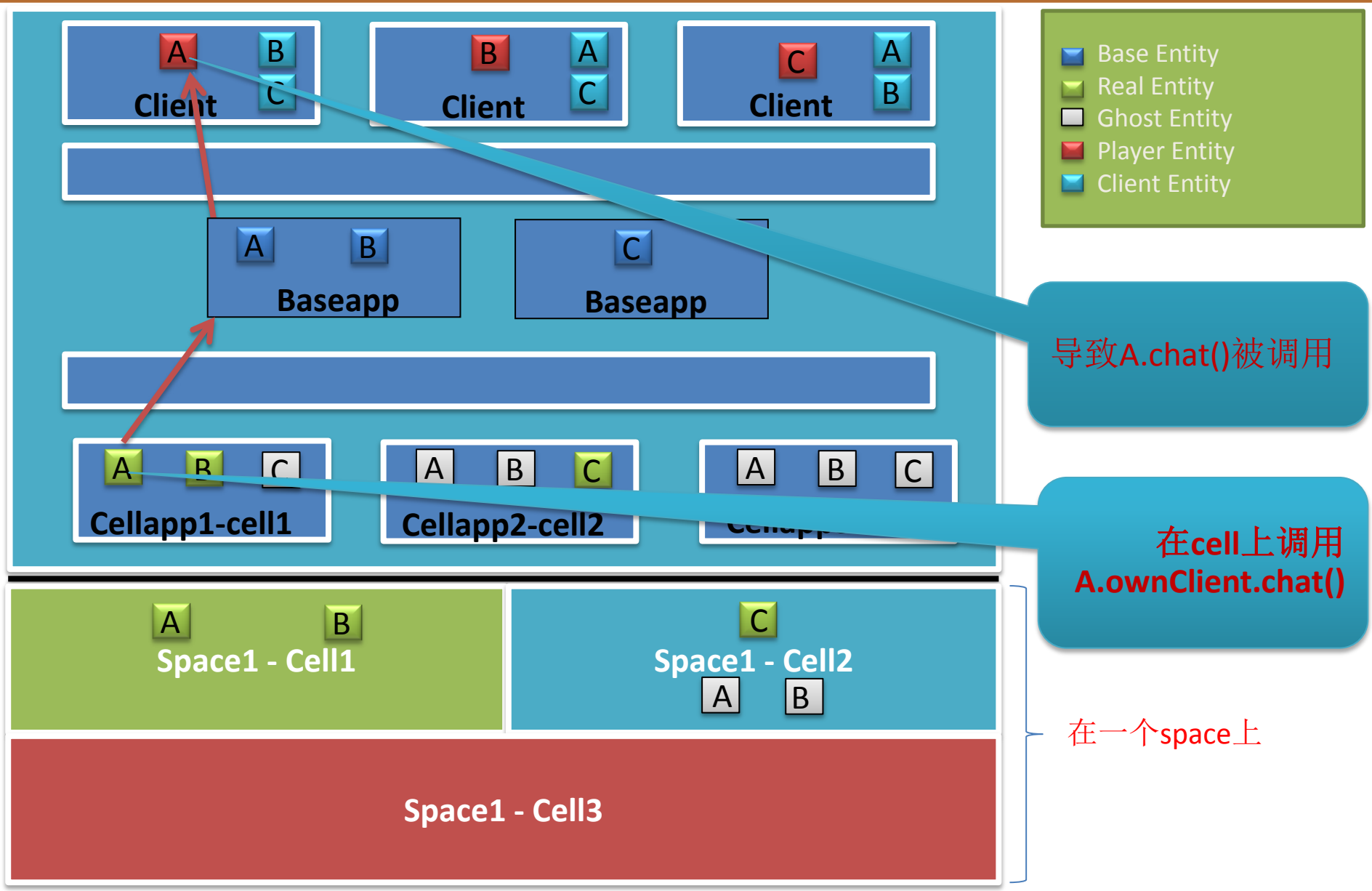


Entity.ownClient 方法调用示例

- self.ownClient.chat() 实际上使得chat函数在A客户端的entity A上被调用。
- 其它的客户端不会意识到A客户端上有A.chat()被调用。

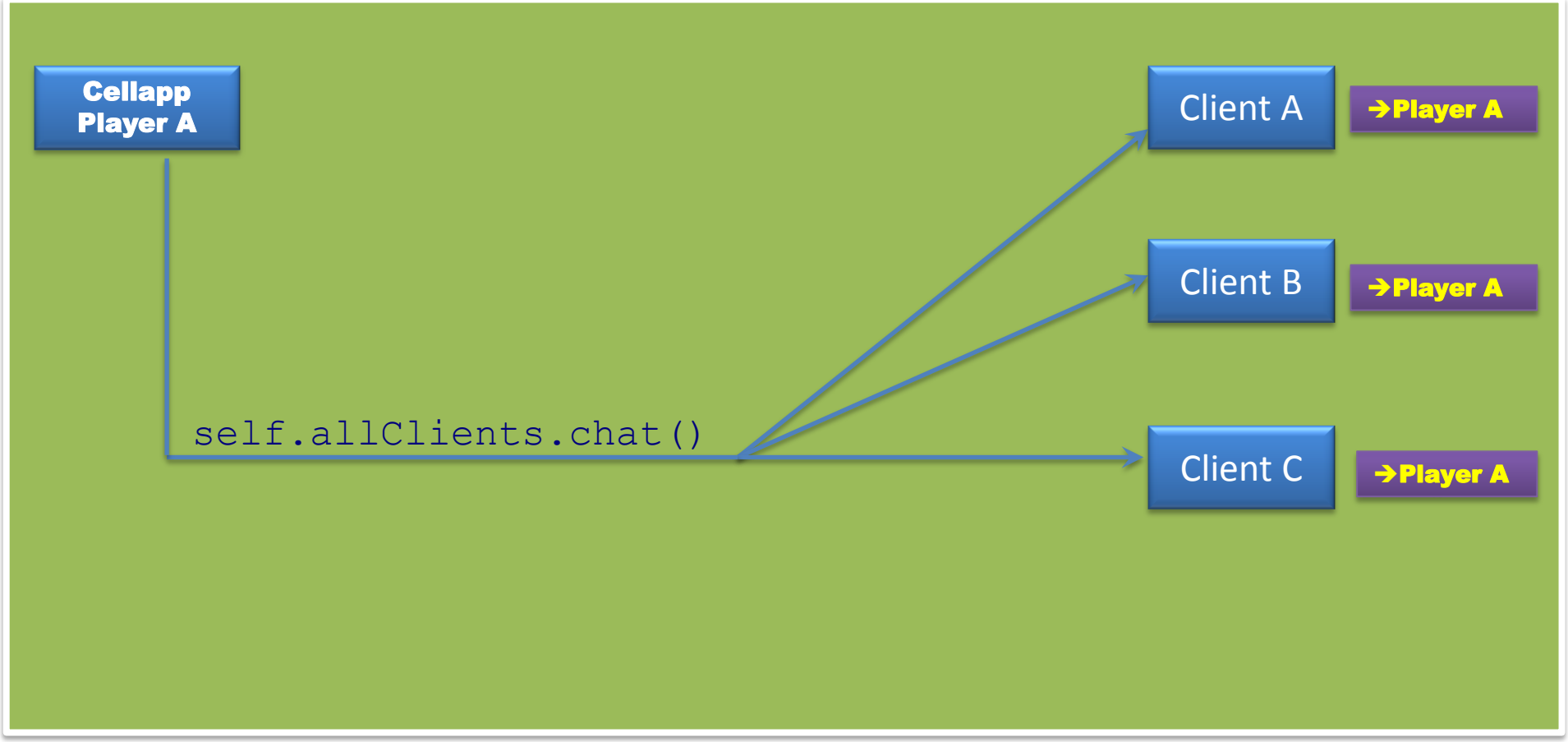


Entity.ownClient 方法调用示例

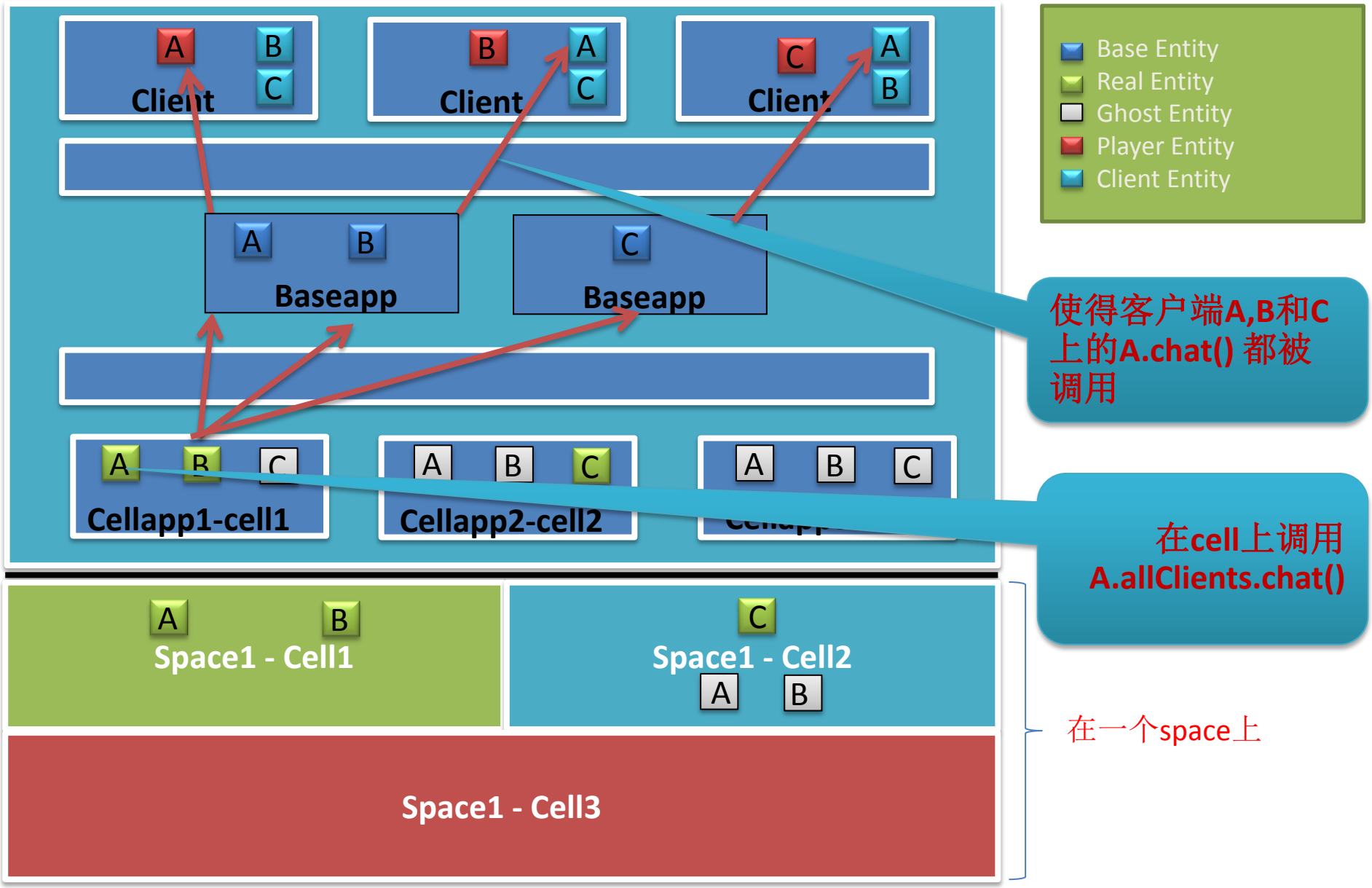


Entity.allClients 方法调用示例

- self.allClients.chat() 使得所有可以看到A的玩家客户端上的entity A的chat()函数被调用。
- 如果一个玩家在A所在的同一个space，并且A处于其AoI范围内，那么这个玩家的客户端就能看到A。



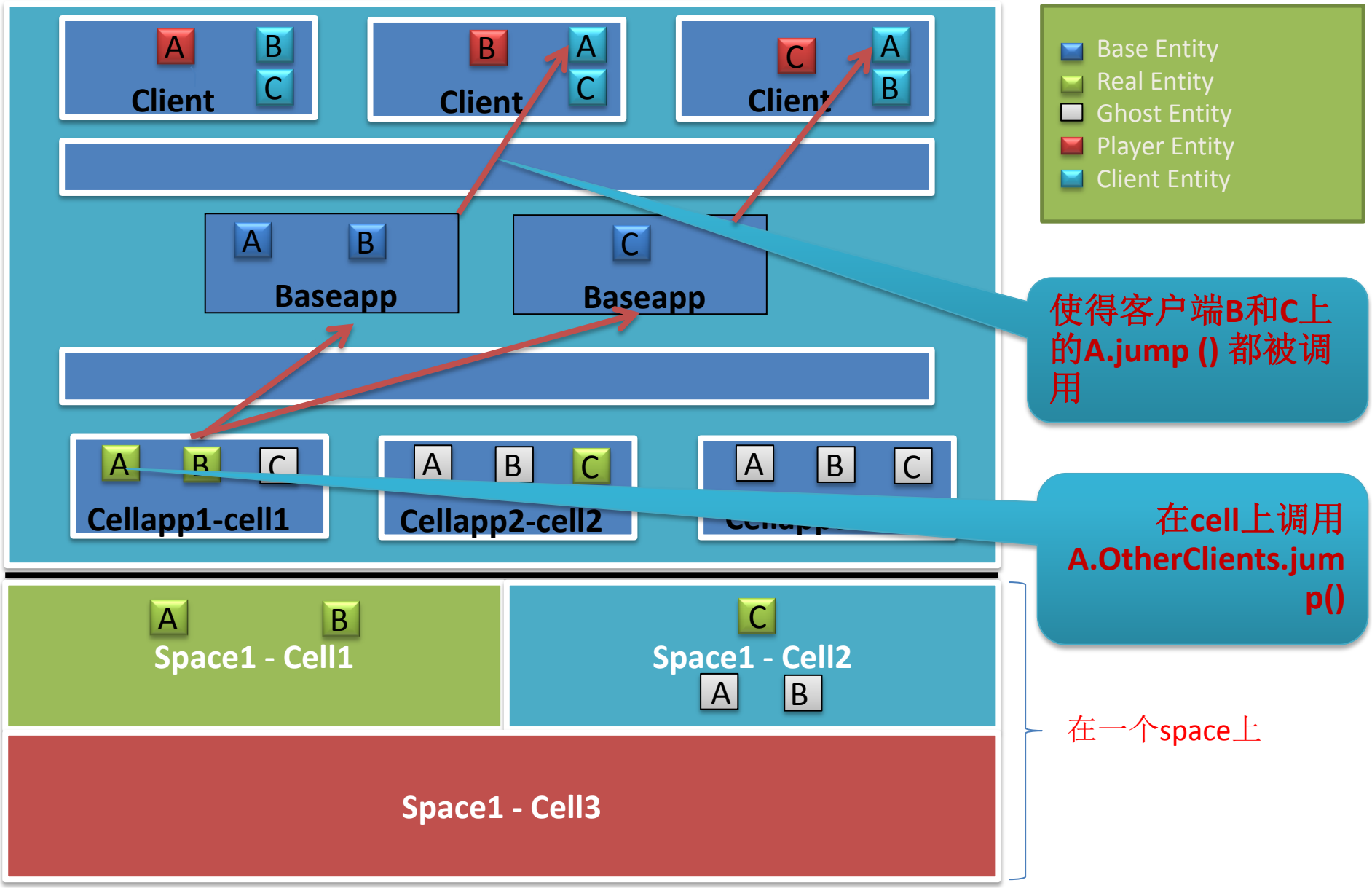
Entity.allClients 方法调用示例



Entity.otherClients 方法调用示例

- self.otherClients.chat()使得所有可以看到A的玩家客户端上的Entity A的chat()函数被调用，A客户端本身除外。
- 如果一个玩家在A所在的同一个Space，并且A处于其AOI范围内，那么这个玩家的客户端就能看到A。
- 通常用于在该玩家客户端立即见到效果的初始动作，它用otherClients的方式把该动作广播到其它玩家客户端。例如：跳跃。

Entity.otherClients 方法调用示例



第四章

核心Entity部件

Baseapp上的Entity类型

- Base
  - 在Python脚本中，继承于**KBEngine.Base**
  - 存放大量复杂的数据
    - 减少当Cell entity跨越Cell边界的时候的系统的负担
  - 是接收方法调用的固定的Mailbox
- Proxy
  - 在Python脚本中，继承于**KBEngine.Proxy**
  - KBEngine.Proxy内部地继承于KBEngine.Base
  - 它是与Client通信的节点
  - Client可以根据需要附加或去除一个Proxy

Baseapp上的Entity属性

- 从KBEngine.Base继承的Entity的属性

属性	描述
id	唯一的Entity ID，Cell,Base,Client共用一个id
databaseID	Entity在数据库里的永久ID。如果不是永久的则为零。64位
cell	如果有对应Cell entity存在，表示指向该Cell entity的mailBox
cellData	如果Cell entity不存在，其以Dictionary结构保留Entity的Cell部分的数据

Baseapp上的Proxy属性

- **KBEngine.Proxy继承自KBEngine.Base，**它是所有有Proxy的Base entity的父类
- 附加属性：

属性	描述
client	用于与对应的客户端的Entity的通信的MailBox
clientAddr	对应客户端机器的地址和端口
bandwidthPerSecond	每秒发送个客户端的信息长度

Baseapp上的Entity方法

方法	描述
addTimer(*initOffset* [,*repeatOffset*, *userData*])	▪增加一个时钟（ offsets为秒数 ），返回它的ID ▪Entity必须实现方法onTimer(*self*, *timerID*, *userData*)
delTimer(*timerID*)	▪删除指定时钟
createCellEntity([*cellMailBox*]) *	▪在Mailbox指向的Cell上创建entity ▪可以用于当在Base上创建Entity时在Cell上初始化一个Cell entity ▪如果不传递CellMailBox，那么Base.cellData[*spaceID*]会被用到
createInNewSpace() *	▪在一个新的space创建一个entity的cell部分（包括一个新的cell来管理它） ▪可以用于创建一个entity来控制一个新的space (*如.*, 任务管理器)
destroyCellEntity()	▪删除Cell entity，保留Base部分 ▪如果是在Space间转移，建议在CellApp上用'teleport'，避免频繁销毁，创建Cell entity ▪此时Base上onLoseCell会被回调，Base.cellData属性会被赋以Cell Entity的属性
destroy()	▪销毁Entity的Base部分 ▪Cell Entity必须已经先被销毁掉 ▪适用于把Entity从游戏中去除 ▪常常被用在onLoseCell回调函数里

* Cell Entity属性被从Base.cellData中取得并传送，而且它变得不再可访问

Cellapp上的Entity属性

属性	描述
id	唯一的Entity id，cell，base，client共用一个id
spaceID	Entity所在的Space
position	Entity的世界坐标系位置
roll	Entity的朝向
pitch	
yaw	
direction	Entity的面向，由roll, pitch, yaw来组合表示
volatileInfo	用来决定roll,pitch,yaw各自的更新频率，在.def文件里有缺省值
topSpeed	Entity的最大速度。用于物理检查

Cellapp上的Entity方法

方法	描述
`destroySpace()`	▪ 删除Space里的所有Entity,从而删除Space
`destroy()`	▪ 删除Entity的Cell部分 ▪ 从Space里删除Entity
`entitiesInRange( range [,entityType, position] )`	▪ 搜索指定范围内的所有Entity ▪ 可以搜索到AOI范围以外的Entity但是无法找到Cell以外的Entity ▪ 球性测试
`isReal()`	▪ 返回此Entity是Real还是Ghost的
`setAoIRadius( radius [, hysteresis] )`	▪ 改变AOI半径，缺省是500m ▪ 必须小于Ghost距离，缺省是500m
`teleport( nearbyEntityMBRef, position, direction )`	▪ 在同一个Space内改变Entity的位置 ▪ 放Entity到另一个Space – `nearbyEntityMBRef`指向的`Entity`相同的`Space`

所有Entity属性和方法都可以在Python API文档内查到:

kbengine/doc/api/kbengine_api.chm

Entity的典型生存周期

- Base部分先被创建
  - 从数据库或者代码里创建
  - Base entity可以没有Cell部分-cellData属性
  - Base entity在其Cell部分存在时不能被销毁
  - Base entity通常在OnLoseCell()回调函数里决定自行销毁
- Cell部分由Base部分来创建
  - Cell-only的entity可以用脚本来创建
- Client部分通常在Entity进入到玩家的AOI时被创建
  - 应该用enterWorld()/leaveWorld()回调函数作为初始和结束，而不是 __init__()函数

Entity的创建

- Entity在Cell上的实例会在下一个网络更新时发布到合适的Client
- 推荐创建的方法:
  - Base Entity:

    `KBEngine.createBaseAnywhere()`
    - 或者:

      `createBaseLocally()`

      `createBase...FromDB()`

Entity的创建

- 推荐的创建方法:
  - Cell Entity:
    `createCellEntity()`
    `createInNewSpace()`
  - Cell entity属性可以在从数据库读取后创建之前修改
    - 参看 Base API 文档: `KBEngine.Base.cellData`
  - Cell Only Entity:
    `createEntity()`
    - 在Cell上调用
    - 不能被容错

Entity的销毁

- Cell entity作为游戏逻辑的一部分被销毁
- Base entity在其Cell部分还存在之前不能被销毁
- 销毁Cell部分:
  - Cell上: `Entity.destroy()`
  - Base上: `Base.destroyCellEntity()`
  - 当`cell`部分销毁时`Base.onLoseCell()`会被调用
- 销毁base部分:
  - `Base.destroy()`
  - 如果是永久性数据会使得`writeToDB()`被调用

容错

- Cell的属性被备份到Base
- Base的属性也被备份到另一个BaseApp
- 永久属性备份到数据库
  - 存档: 持续地轮流调度存档
- 容错vs.灾难恢复
  - 灾难 = 同时很多服务器进程失败

第五章

Cell功能集

Entity的cell部分的功能集

- 在Entity的Cell部分有许多与空间有关的功能可用
  - Navigate/MoveTo*导航系统
  - Proximity触发器(陷阱)
- 这些功能都是用Controller实现的

Entity的Controller

- 实现那些需要在后台花费很多tick处理的功能
- 当结束时会回调Python脚本
- 用于实现复杂的逻辑
- 因为效率原因用C/C++实现(相对于script)
- 当Entity跨越Cell边界时Controller也跟着复制到新的Cell
- 每个Entity上可以有无限多个Controller
- 每个实列返回一个Controller ID
  - 删除:`Entity.cancel( id )`
- 能在它们的Entity的脚本上激活回调函数

Entity的导航系统(Navigation)

- 导航系统提供了许多用于Entity的移动和寻路的函数
- Navigation Controller用RecastNavigation从静态的碰撞场景里预先产生的NavMesh来寻路

Entity的导航系统(方法)

- 直接的直线运动
  - `moveToPoint()`
  - `moveToEntity()`
- 导航(用NavMesh)
  - `navigate()`
- 通常的
  - `canNavigateTo()`
  - `getStopPoint()`

Entity Proximity

- ProximityController实现一个无限高的，与轴平行的立方柱形的陷阱
- 应该在陷阱的通知函数中再进行Y轴的检查
- 一个Entity可以有很多个Proximity陷阱
- 增加一个Proximity陷阱:
  Entity.addProximity()

控制其它的Entity

- 包括2个部分:
  - 客户端发送位置更新到新的Entity: `KBEngine.controlEntity()`
  - 服务器接受Entity的位置更新: `Entity.controlledBy`
    - 设置成控制该Entity的玩家的Mailbox
- 这个Entity不能超过控制玩家的AOI范围之外 (Proxy Entity)
  - 因此, 基本上仅适合于玩家坐骑的vehicle
- 或者, 可以从一个玩家转移控制到另一个玩家 (两者都应该有Proxy base部分)
  - `Proxy.giveClientTo()`
    - `Entity.controlledBy` 会自动地设置给新的玩家
  - 分裂型的 – AOI被销毁, 重建， Space被重新加载

第六章

KBEngine服务器设定

服务器配置

- **kbengine.xml** – Server配置文件
  - 指定许多server运行时刻的参数
  - 在server资源路径下
  - 完整的文档见http://www.kbengine.org/docs/configuration/kbengine.html
- Personality个性化脚本
  - 实现全局的回调函数
  - 用KBEngine Python接口处理系统级的消息事件
    - 例如: 启动, 恢复, 关闭
  - 可以理解为入口(在服务器启动后，服务器准备好了的回调里开始构建世界)
  - 缺省情况下Cellapp和Baseapp脚本是分离的(cell/kbengine.py, base/kbengine.py)
  - Personality脚本名在**kbengine.xml**文件里指定。缺省是kbengine

Personality个性化脚本

- Cellapp Personality脚本可以在 **onCellAppReady**时设定游戏
  - 用import KBEngine来使用KBEngine函数
  - **KBEngine.addSpaceGeometryMapping(self.spaceID, None, "spaces/demo")**
    - 参考API文档
- Baseapp Personality脚本可以在 **onBaseAppReady**时设置游戏
  - 如果要创建全局base的话，可以在这个时候创建
  - 应该在这里创建新的space
- 以上两个脚本必须都必须执行清理工作：
  - 在**onBaseAppShuttingDown** 或 **onCellAppShuttingDown**被调用的时候
  - Baseapps同时还在接近结束的时候接收到**onBaseAppShutDown**消息
- Personality脚本可以根据需要执行其它的任务
  - 是置放全局游戏脚本的地方，但不要把所有东西都放在里面
  - 对每个逻辑部分用分开的脚本文件

第七章

服务器调试

C++断点调试

- 尽可能的使用Log追踪执行过程
- 服务端进程断点请启动完服务组后附加到进程

特殊情况请设置好系统环境变量，先启动好依赖进程之后使用IDE单独启动进程调试

工具与服务端交互调试

- 使用GUIConsole-Debug页能够在内存中与Cellapp或Baseapp的Python脚本交互
- 使用kbengine/kbe/tools/server/pycluster/cluster_controller.py或命令能够telnet到服务端与Cellapp或Baseapp的Python脚本交互
- 用KBEngine Python接口来交互
  - 例如：在Baseapp上
  - `>>> e = KBEngine.createBase( "SpawnPoint", position = (2, 3, 5) )`
  - `>>> e.id`
  - `1234`

  - 例如：在Cellapp上:
    - `>>> e = KBEngine.entities[实体的ID]`
    - `>>> e.position`
    - (1.000000, 2.000000, 3.000000)
      - 注意y是在KBEngine里的竖直高度
    - `dir(e)`
      - 可以查看许多内建的属性，方法。还有在entity定义里的entity特定的属性和方法
    - `e.destroy()`
      - 使得Entity base能够销毁自己

更多参考: http://www.kbengine.org/docs/documentations/onlinedebugging.html

第八章

Profiling和压力测试

用机器人做压力测试

- 模拟大量的玩家
- 强烈建议在大规模玩家测试前进行压力测试
- 不要有地形的加载
- 不要有导航系统
- 和空间有关的游戏不要大量聚集到一个小范围

Bot脚本

- 每个类型的Entity在**<assets>/scripts/bot**下面都需要一个Python脚本
  - Bot脚本应该实现Entity的Client部分
    - 但是因为bots脚本没有许多Client里用到的UI和3D的部分，所以简单的复制client脚本是不行的
  - 对大多数Entity类型，实现一个空的class就可以
  - 对Account和Player entity，需要编写登录的脚本和模拟玩家的脚本
  - 编写A.I.来模拟一个玩家

增加bots

- 运行bot进程再运用GUIConsole来增加bots
- 或者在 **kbengine/kbe/res/server/kbengine_defs.xml/bots** 中设置机器人初始数量和自增到最大数量的控制

Profiling 工具

- GUIConsole 的profile页面有很多可以用来profile一个运行的服务器群组的各个方面的指数(仅支持Windows)
- 使用kbengine/kbe/tools/server/pycluster/cluster_controller.py也可以在命令行使用命令profile
- Graphs 可以为你显示每个server进程的负载
- 请注意应该尽早的profiling，注意你的内部的带宽和外部的带宽不会被复杂的方法调用占光了
  - 同时推荐使用单独的网络硬件来用于监视工具，这样可以准确的判断出什么时候网络饱和了
- 使用profiling得到的数据来定位需要优化的部分

Profiling命令

- eventprofile – 诊断出消耗最大的方法调用和状态更新。
- networkprofile – 诊断出占用带宽最大的消息。
- pyprofile – 诊断出消耗cpu时间最多的python函数调用。
- cprofile – 诊断出消耗cpu时间最多的引擎的c++函数调用。

更多参考

- https://github.com/kbengine/kbengine_docs
- http://www.kbengine.org/docs/