Package 'methods'

December 19, 2016

Version 3.3.2

Priority base

Imports utils, stats

Title Formal Methods and Classes

Author R Core Team

Maintainer R Core Team <R-core@r-project.org>

Description Formally defined methods and classes for R objects,
plus other programming tools, as described in the reference.

References John M. Chambers (2008)
``Software for Data Analysis: Programming with R''; Springer NY.

License Part of R 3.3.2

Suggests codetools

NeedsCompilation yes

R topics documented:

`methods-package` 形式的メソッドとクラス

Description

R オブジェクトに対して形式的に定義されたメソッドとクラス，に加えて参考文献に説明された他のプログラミングツール．

Details

このパッケージは関数言語におけるメソッドとクラスに対する 'S4' または 'S バージョン 4' アプローチを提供する．

これらのトピックスに対するかなり技術的なレベルの一般的議論については `Classes`, `Methods` そして `GenericFunctions` のドキュメント項目を見よ．このページからのリンクと `setClass` と `setMethod` のドキュメントは必要なプログラミングツールをカバーする．

関数とクラスの完全なリストには `library(help="methods")` を使う．

Author(s)

R コアチーム

管理者：R Core Team <R-core@r-project.org>

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

`.BasicFunsList` 組み込みと特殊関数

Description

組み込みと特殊関数を総称的関数に変換する指示を提供する名前付きリスト．

`.Primitive(<name>)` として定義された R 中の関数は，基本的な反射特性を持たないため，形式的メソッドに対しては適当でない．これらの関数に対する引数リストは関数オブジェクト自体を調べることでは見つけることが出来ない．

将来のバージョンの R はこれを対応する関数に形式的引数リストを付加することでフィックスするかもしれない．一般的に引数の名前は関数を実装する内部コードでチェックされないが，幾つかの引数は頻繁にそうされる．

どの場合でも，もしユーザがこれらの関数に対するメソッドを定義しようとすると形式的引数のある種の定義が必要になる．特に，もし複数のパッケージからのメソッドが混

用されるならば，メソッドの異なったセットは形式的引数に関して一致する必要がある．

反射性が無ければ，このリストはそれに対してメソッドが許される既知の特殊関数の各々に対して関連するダミー関数を用いて関連する情報を提供する．

同時に，リストはメソッドが意味を持たない(例えば for) か単にまずいアイデアである(例えば .Primitive)特殊関数にフラグを立てる．

例えば， setMethod により作られるこれらの特殊関数の一つに対する総称的関数は .BasicFunsList からの引数リストを持つ．もし何も項目がなければ，引数リスト (x, ...) が仮定される．

---

as　　オブジェクトがあるクラスに属することを強制する

Description

これらの関数はオブジェクトを与えられたクラスに強制変換することを許す関係を管理する．

Usage

```
as(object, Class, strict=TRUE, ext)

as(object, Class) <- value

setAs(from, to, def, replace, where = topenv(parent.frame()))
```

Arguments

object	任意の R オブジェクト．
Class	object が強制変換されるべきクラス名．
strict	論理フラグ．もし TRUE ならば返されるオブジェクトは厳密に目標クラスからのものでなければならない (クラスが仮想的でない限り，その場合はオブジェクトは最も近い実際のクラスからのものになる．特にもしそのクラスが仮想的クラスを直接拡張していればオリジナルのオブジェクトになる)． もし strict = FALSE ならば，目標クラスの任意の単純な拡張がそれ以上の変更無しで返される．単純な拡張とは簡単に言えば既存クラスに単にスロットを加えたものである．
value	object を修正するのに使われる値 (下の議論を見よ)．クラス Class のオブジェクトを提供すべきである；しかしそれを過信するのは賢明ではない．
from, to	強制変換メソッド def と replace がその間で強制変換を実行するクラス．
def	引数が一つの関数．これはクラス from からのオブジェクトを受け取りクラス to のオブジェクトを返すことが好ましい．変換は引数名が from であるようなものである；もし別の引数名が使われると， setAs は from を代入しようと試みる．

replace
: もし提供されると， as が付値の左辺で使われる際の置き換えメソッドとして使われる関数．二つの引数 from，value の関数でなければならないが， setAs はもし引数が異なれば代入しようと試みる．

where
: 結果のメソッドをその中に保存する位置もしくは環境．殆どの応用に対してこの引数を省略し，ソースコード中にトップレベルで評価される setAs への呼び出しを含めることが勧められる；つまり，R セッション中では source への呼び出しに等しい何かか，またはパッケージに対する R ソースコードの一部分として．

ext
: どのように Class がオブジェクトのクラスで拡張されるかを定義するオプションのオブジェクト (possibleExtends が返すような)．この引数は内部的に使われる (非公開クラスに対する本質的情報を提供するために)が，それを直接使いたくなることは恐らく無い．

関数の要約

as: このオブジェクトを与えられた Class に強制変換したバージョンを返す．付値の左辺側からの置き換え中で使われた時はオブジェクトの Class に対応する部分が value で置き換えられる．

いずれかの形式の as() の操作は強制変換メソッドの定義に依存する．メソッドは二つのクラスが継承で関係付けられているときは自動的に定義される；つまりクラスの一方が他方のサブクラスである時．詳細は下の継承の節を見よ．

強制変換メソッドはまた基本クラスに対して予め定義されている (全てのベクトルタイプ，関数そして幾つかの他のもの)．これらの一覧には showMethods(coerce) を見よ．

これらのメソッドの二つのソースを超えて，更なるメソッドが setAs 関数の呼び出しにより定義されている．

setAs: クラス from のオブジェクトをクラス to へ強制変換するためのメソッドを定義する；引数 def は直接的な強制変換を提供し，もしあれば replace 引数は置き換え用を定義する．詳細は下の“関数 'as' と 'setAs' がどのように動作するか”節を見よ．

coerce, coerce<-: from を to と同じクラスに強制変換する．

これらの関数は明示的に呼び出されるべきではない．関数 setAs はそれらに対するメソッドを使用する as 関数のために作る．

継承と強制変換

オブジェクトは一つのクラスから別のクラスへのオブジェクトへ自動的にか as 関数への明示的な呼び出しで変換できる．自動変換は特殊であり，このオブジェクトのクラスが別のクラスを拡張するというデザイナーの保証に基づく．最も普通のケースは一つまたはそれ以上のクラスが setClass への contains= 引数中に提供されることで，その場合新しいクラスはそれ以前のクラスの各々を拡張する (普通の言い回しでは以前のクラスは新しいクラスの *superclasses* でありそれらの各々の *subclass* である)．

この継承の形式は R の単純な継承と呼ばれる．詳細は setClass を見よ．継承はまた setIs の呼び出しで明示的に定義できる．二つのバージョンは強制変換に対して僅かに異なる含意を持つ．単純な継承は継承スロットがサブクラスとスーパークラスで同じように動作することを意味する．二つのクラスが単純な継承で結びついている場合は常に対応する強制変換メソッドが直接と as の置き換え使用の双方に対して定義される．単純な継承のケースではこれらのメソッドは明白な計算を行う：それらはスーパークラス定義中のそれらに対応するオブジェクト中のスロットを取り出したり置き換えたりする．

暗黙のうちに定義された強制変換メソッドは setAs への呼び出しで書き換えることが出来る；しかしながら暗黙のメソッドは各々のサブクラス-スーパークラスの対に対して定

義され，従ってこれらの各々を継承に依るのではなく明示的に書き換える必要があることを注意する．

継承が setIs への呼び出しで定義されるときは，強制変換メソッドは明示的に提供され，自動的には生成されない．継承は(下の節で解説されるように from 引数に)適用される．非継承的な関係に対するメソッドも setAs を使って提供することも可能ではあるが，その時はまたこれらは継承することが出来る．

単純と明示的な継承の間の差異に関するこれ以上については setIs を見よ．

関数 'as' と 'setAs' がどのように動作するか

関数 as は object をクラス Class のオブジェクトに変換する．そのために S4 クラスとメソッドを使う"強制変換メソッド"を適用するが，幾らか特別な仕方でである．強制変換メソッドは関数 coerce に対するメソッドであるか，置き換えケースでは関数 `coerce<-` である．これらの関数はメソッドシグネチャ中の二つの引数 from と to を持ち，オブジェクトのクラスと希望の強制変換先のクラスに対応する．これらの関数は直接呼び出してはならないが，直接または置き換えに対して as を使用されるメソッドのテーブルを保管するのに使われる．この節では直接的なケースを解説するが，注意がある箇所を除けば置き換えケースも coerce と the def 引数の代わりに `coerce<-` と setAs への replace 引数を用いて同じ仕方で動作する．

object がすでに希望のクラスでないことを仮定すると， as は先ずシグネチャ c(from = class(object), to = Class) に対する関数 coerce に対するメソッドのテーブル中のメソッドをメソッド選択がその最初の探索で行うのと同じ方法で探す．正確に言うと，これは直接と継承メソッドの双方のテーブルを意味するが，継承はこの場合は特殊である (下を見よ).

もし如何なるメソッドも見つからなければ as は何かを探す．最初に，もし Class か class(object) が他方のスーパークラスならば，クラス定義は強制変換メソッドを作るのに必要な情報を含む．サブクラスがスーパークラスを含む普通のケース (つまり全てのそのスロットを含む)では，メソッドは継承スロットを取り出すか置き換える．単純でない拡張(setIs への呼び出しの結果)は通常明示的なメソッドを含むが，置き換えに対しては含まないかもしれない．

もしサブクラス/スーパークラス関係がメソッドを提供しなければ as は継承メソッドを探すが引数 from に対してだけで引数 to に対しては適用されない (結果が指定された Class 以外の他のクラスからのものになることは好ましくないことに恐らく同意するであろう)．従って selectMethod("coerce", sig, useInherited= c(from=TRUE, to= FALSE)) as() で使われるメソッド選択を複製する．

ほとんどすべてのケースでこのようにして見つけられたメソッドは強制変換メソッそのテーブル中にキャッシュされる (例外は検査を備えたサブクラス関係で，これは適正であるが勧められない)．従って詳細な計算は class(object) から Class への強制変換が最初に起きた時だけなされるべきである．

coerce は標準的な総称的関数では無いことを注意する．これは直接呼び出されることを意図していない．間違って不正な継承メソッドをキャッシュすることを防ぐために，呼び出しは as への同値な呼び出しに振り向けられる．またこの関数に対する selectMethod への呼び出しは as が選ぶメソッドを表していないかもしれない．もし対応する as への呼び出しがこのセッションで先に起きている時だけ結果を信頼できる．

この説明を背景として，関数 setAs はかなりわかりやすい計算を行う：それはメソッドの本体を定義する def 引数を用い，シグネチャ c(from, to) を持つ関数 coerce に対するメソッドを構成し設定する． def として提供された関数は引数を一つ持つか(強制変換されるオブジェクトと解釈される) または二つの引数 (from オブジェクトと to クラス)を持つことが出来る．どちらの場合も， setAs は二つの引数の関数の関数を構成し，二つ目は to クラスの名前が既定値になる．メソッドはオブジェクトを from 引数とし to 引数を

持たずに as から呼び出され，この引数に対する既定値は意図された to クラスの名前であり，従ってメソッドはメッセージ中でこの情報を使うことが出来る．

as 関数の直接バージョンはまた既定値が TRUE の引数 strict= を持つ．他の関数に対するメソッドの評価の間の呼び出しはこの引数を FALSE に設定する．この違いは強制変換されるオブジェクトが to クラスの単純なサブクラスからのものであるときは意味を持つ：この場合もし strict=FALSE ならば何もする必要はない．殆どのユーザーが書いた強制変換メソッドに対しては，二つのクラスがサブクラス/スーパークラスを持たない時は， strict= 引数は無関係である．

setAs への replace 引数は `coerce<-` に対するメソッドを提供する．全ての置き換えメソッドの場合と同様に，メソッドの最後の引数は付値の右辺値に対する名前 value を持たなければならない． coerce メソッドの場合と同様に，最初の二つの引数は from，to である；置き換えケースには strict= オプションは無い．

関数 coerce はそうしたメソッドに対する保管庫として存在し， as 関数により上で説明されたように選択される．実際には standardGeneric を用いたメソッドの選択適用は to 引数に関する継承を用い不正確な継承メソッドを作り出すかもしれない；説明されたようにこれは as に対して用いられるロジックではない．不正なメソッドを選択しキャッシュ防ぐために， coerce への呼び出しは現在警告と共に as への呼び出しにマップされる．

基本的な強制変換メソッド

任意のオブジェクトを基本的なデータタイプの一つに強制変換するためのメソッドが予め定義されている．例えば as(x, "numeric") は既存の as.numeric 関数を使う． これらの組み込みメソッドは showMethods("coerce") で一覧できる．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

もし try(as(x, cl)) を使おうと思うならば代わりに canCoerce(x, cl) を考慮しよう．

Examples

```
## \link{setClass} からのクラス "track" の定義を使う



setAs("track", "numeric", function(from) from@y)

t1 <- new("track", x=1:20, y=(1:20)^2)

as(t1, "numeric")

## 次の例は以下を示す：
##  1. 幾つかのクラスに対して setAs を同時に定義する仮想クラス．
##  2. 継承情報を使う as()

setClass("ca", representation(a = "character", id = "numeric"))
```

```
setClass("cb", representation(b = "character", id = "numeric"))

setClass("id")
setIs("ca", "id")
setIs("cb", "id")


setAs("id", "numeric", function(from) from@id)

CA <- new("ca", a = "A", id = 1)
CB <- new("cb", b = "B", id = 2)

setAs("cb", "ca", function(from, to )new(to, a=from@b, id = from@id))

as(CB, "numeric")
```

BasicClasses 基本データタイプに対応するクラス

Description

R 基本オブジェクトタイプに対応する形式的クラスが存在し，クラス定義中のスロットとしてメソッドのシグネチャ中でこれらのタイプを使ったり，新しいクラスで拡張したりすることを許す.

Usage

```
### 以下は全ての基本ベクトルクラスである.
### これらはメソッドのシグネチャ中や as(), is()
### そして new() への呼び出しでクラス名として登場出来る.
.
"character"
"complex"
"double"
"expression"
"integer"
"list"
"logical"
"numeric"
"single"
"raw"

### クラス
"vector"
### は仮想的なクラスで，上の全てにより拡張される.

### クラス
"S4"
### はクラス S4 のオブジェクトで基本的なベクトルクラスの
```

```
### どれも拡張しない．これは仮想的クラスである．

### 次は追加の基本的クラスである
"NULL"      #  NULL オブジェクト
"function" #  プリミティブを含む関数オブジェクト
"externalptr" # C コードで使うバイト型外部ポインタ

"ANY"  # メソッドのパッケージ自体で使われる仮想的クラス
"VIRTUAL"
"missing"

"namedList" # 名前属性を保存する "list" の代替物
```

クラスからのオブジェクト

クラスが仮想的で無ければ(Classes 中の節を見よ)，オブジェクトは形式 new(Class, ...) の呼び出しで作ることが出来，ここで Class は引用符付きのクラス名で，残りの引数はもしあればこのクラスのベクトルとして解釈されるオブジェクトである．複数の引数は連結される．

クラス "expression" は ... 引数が評価されないという点で少々奇妙である；従ってそれらを quote() への呼び出しで囲まない．

クラス "list" は純粋なベクトルであることを注意する．名前付きリストは S の最も初期のバージョンに遡るが，それらは属性(今はスロットで良い)を持つという点でベクトルの概念の拡張であり，属性は NULL かベクトルと同じ長さの文字列ベクトルである．もしリスト名が保存されることを保証したければ "list" ではなくクラス "namedList" を使う．このクラスからのオブジェクトはスロット "names" に対応しタイプ "character" の名前属性を持たなければならない．内部的には R はリストに対する名前を特別に扱い，クラス "namedList" 中の対応するスロットが文字列名と NULL の合併であることは実際的ではない．

クラスとタイプ

基本クラスは基本的な R タイプに対するクラスを含む．これらのタイプは普通 S4 オブジェクトではない (isS4 は FALSE を返す)が，基本クラスを含むクラスからのオブジェクトは S4 オブジェクトであり，依然同じクラスである． typeof が返すようなタイプは時おりクラスが異なるが，単に用語の選択からか (例えば，タイプ "symbol" とクラス "name")またはクラスとタイプ間に一対一対応が無いせいである (例えば，クラス "language" を継承する殆どのクラスはタイプ "language" を持つ).

拡張

ベクトルクラスは "vector" を直接に拡張する．

メソッド

coerce 対応する基本関数を呼び出すことにより，任意のオブジェクトをベクトルクラスに強制変換するメソッドが定義されている，例えば as(x, "numeric") は as.numeric(x) を呼び出す．

callGeneric　　メソッドから現在の総称的関数を呼び出す

Description

callGeneric の呼び出しはメソッド定義の内部にだけ現れることが出来る．そうするとそれは現在の総称的関数への呼び出しを行う．この呼び出しの結果は callGeneric の値である．これは任意のメソッドから呼び出すことができるが，グループ総称的関数に対するメソッド中で有用であり典型的に使われる．

Usage

```
callGeneric(...)
```

Arguments

...	オプションで，その次の呼び出し中の関数への引数． callGeneric への呼び出し中に如何なる引数も含まれなければ，その効果は関数を現在の引数で呼び出すことである．これが実際に意味することは詳細説明を見よ．

Details

現在の総称的関数の名前とパッケージはメソッド定義オブジェクトの環境に保管される．この名前は検索され対応する関数が呼び出される．

callGeneric へ引数を渡さない文は総称的関数を現在の引数で呼び出すことはより正確には次のようになる．現在の呼び出し中に欠損していた引数は依然として欠損している ("missing" はメソッドシグネチャ中の適正なクラスであることを思い出そう)．オリジナルの呼び出し中に現れる形式的引数，例えば x に対しては x = x に同値な対応する引数が生成された呼び出し中にある．実際には，これは総称的関数が同じ実際の引数を見るが引数は唯一回だけ評価されることを意味する．

引数無しで callGeneric を使うことは，シグネチャ中の引数の一つが現在のメソッド中で変更されており従って異なったメソッドが選択されない限り，無限の再帰を作りやすい．

Value

新しい呼び出しが返す値．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

GroupGenericFunctions for other information about group generic functions; Methods for the general behavior of method dispatch

Examples

```
## グループ総称的関数 Ops に対するシグネチャ
## signature( e1="structure", e2="vector") のメソッド
function (e1, e2)
{
    value <- callGeneric(e1@.Data, e2)
    if (length(value) == length(e1)) {
        e1@.Data <- value
        e1
    }
    else value
}

## もっと多くの例には
## Not run:
showMethods("Ops", includeDefs = TRUE)

## End(Not run)
```

---

callNextMethod　　継承メソッドの呼び出し

Description

callNextMethod の呼び出しはメソッド定義の内部にだけ登場できる．するとそれは現在のメソッドの最初の継承メソッドへの呼び出しになり，現在のメソッドへの引数は次のメソッドに引き渡される．そのメソッド呼び出しの値は callNextMethod の値である．

Usage

```
callNextMethod(...)
```

Arguments

...	オプションのその次の呼び出し中の関数への引数 (しかし選択適用は下で説明されるようなものであることを注意する；この引数は次のメソッドの選択に対しては何の影響も持たない．) もし callNextMethod への呼び出し中に何の引数もなければ，その効果はメソッドを現在の引数で呼び出すことである．これが実際に意味するところについては詳しい解説を見よ． 引数無しの呼び出しはしばしば callNextMethod を使う自然な方法である；例を見よ．

Details

‘次’のメソッド(つまり最初の継承メソッド)は，もし現在のメソッドが存在しなかった時に呼び出されたであろうメソッドと定義される．これは多かれ少なかれ文字通り起こることである：現在のメソッド(正確にはそれから callNextMethod が呼び出されるメソッドの defined スロットによって与えられるシグネチャを持つメソッド) は現在の総称的関数に対するメソッドのコピーから削除され，そして selectMethod が次のメソッドを見

つけるために呼び出される (結果は特殊なオブジェクト中にキャッシュされるので，検索は典型的にセッション毎，引数のクラスの組み合わせ毎にだけ起こる).

先の定義は setMethod がシグネチャ中のクラスを与えて callNextMethod 呼び出しを含むメソッドを挿入する時一意的に次のメソッドが定義されることを意味する．選択はそのメソッドに導くパスに依存しない (例えば継承を通じてか別の callNextMethod 呼び出しから)．この定義はバージョン2.3.0以前の R では強要されず，そこではメソッドはターゲットのシグネチャに基づいて選択されたので実際の引数に応じて変わることがあり得た.

callNextMethod により呼びだされたメソッドがそれ自体 callNextMethod への呼び出しを持つことは適正であり，そしてしばしば有用である．これは一般に期待されたように動作するが，完全さのためには，二つの他のクラス定義中に同じ二つのクラスが 反対の順序のスーパークラスとして登場するかもしれないという意味で，S 構造中に曖昧な継承が存在する可能性があることを注意する．この場合入れ子の callNextMethod の効果は良くは定義されていない．そうした一貫性のないクラス継承は稀であると共にほとんど常にまずいデザインの結果であるが，可能ではあり現在検出されない.

現在の引数でメソッドが呼び出されるということの意味はより正確には次のようになる．現在の呼び出しで欠損している引数は依然欠損する ("missing" はメソッドのシグネチャ中の適正なクラスであることを思い出そう)．オリジナルの呼び出しに登場した形式的引数，例えば x，に対しては次のメソッド呼び出しに x = x と同値な対応する引数がある．効果としては，次のメソッドは同じ実際の引数を持つが，引数は一度だけ評価される.

Value

選択されたメソッドによる返り値.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

現在の選択適用規則を持つ総称的関数を呼び出すには callGeneric (典型的にはグループ総称的関数に対して)；メソッドの選択適用の一般的挙動については Methods.

Examples

```
## 単純な継承を持つあるクラス定義
setClass("B0" , representation(b0 = "numeric"))

setClass("B1", representation(b1 = "character"), contains = "B0")

setClass("B2", representation(b2 = "logical"), contains = "B1")

## そして callNextMethod を説明するかなり馬鹿げた関数

f <- function(x) class(x)

setMethod("f", "B0", function(x) c(x@b0^2, callNextMethod()))
setMethod("f", "B1", function(x) c(paste(x@b1,":"), callNextMethod()))
```

```
setMethod("f", "B2", function(x) c(x@b2, callNextMethod()))

b1 <- new("B1", b0 = 2, b1 = "Testing")

b2 <- new("B2", b2 = FALSE, b1 = "More testing", b0 = 10)

f(b2)
stopifnot(identical(f(b2), c(b2@b2, paste(b2@b1,":"), b2@b0^2, "B2")))

f(b1)

## よりa sneakier method: *変更された* x を使う：
setMethod("f", "B2",
          function(x) {x@b0 <- 111; c(x@b2, callNextMethod())})
f(b2)
stopifnot(identical(f(b2), c(b2@b2, paste(b2@b1,":"), 111^2, "B2")))
```

---

canCoerce　　オブジェクトをある *S4* クラスに強制変換できるか?

Description

あるオブジェクトが与えられた S4 クラスに強制変換できるかどうかをテストする. if() の内部で呼び出し as(object, Class) がメソッドを見つけられるかどうかを確認するのに役立つかもしれない.

Usage

```
canCoerce(object, Class)
```

Arguments

object	任意の R オブジェクトで，典型的に形式的 S4 クラス.
Class	ある S4 クラス(isClass を見よ).

Value

スカラの論理値，もし (from = class(object), to = Class) に対する coerce メソッド (例えば setAs で定義されるような)があれば TRUE.

See Also

as, setAs, selectMethod, setClass,

Examples

```
m <- matrix(pi, 2,3)
canCoerce(m, "numeric") # TRUE
canCoerce(m, "array")   # TRUE
```

cbind2 二つのオブジェクトを列や行で連結する

Description

二つの行列風 R オブジェクトを列(cbind2) か行(rbind2)で連結する．これらは既定メソッドを持つ(S4)総称的関数である．

Usage

```
cbind2(x, y, ...)
rbind2(x, y, ...)
```

Arguments

x	任意の R オブジェクトで，典型的には行列風.
y	任意の R オブジェクトで，典型的には x に類似か完全に欠損.
...	メソッドに対するオプションの引数.

Details

cbind2 (rbind2) の主な使い道は cbind() (rbind()) から以下の要請下で再帰的に呼び出されることである：

- 少なくとも一つの S4 オブジェクトがあり，そして
- S3 選択適用が失敗する (cbind の選択適用の節を見よ).

cbind2 と rbind2 へのメソッドは非等質的な引数のセットを連結する際タイプの奨励手順を実効的に定義する．全てのオブジェクトがある S4 クラスから導かれている等質的なケースは外部的に定義された S4 cbind (rbind) 総称的関数を使った ... 引数への S4 選択適用を使って処理される．

(遺物的な理由から) S3 選択適用が最初に試みられるので， S4 クラスに対する cbind (rbind) への S3 メソッドを追加で定義することは一般に良いアイデアである． S3 メソッドは，S3 メソッドが存在しないようなクラスの引数と並んで，引数が S4 クラスのオブジェクトを含む時に起動される．同様に，異なった S3 メソッドを選択する引数がある場合には (data.frame に対するもののように)，この S3 メソッドは cbind2 (rbind2) への再帰的なフォールバックを惹起する選択適用中に曖昧さを導入する働きをする．さもなければ，他の S3 メソッドが呼び出され，それは適切ではないかもしれない．

Value

x と y の列(または行)を連結した行列(または行列風オブジェクト)．メソッドは対応する x と y の対応する列名と行名から colnames と rownames を作らなければならない (しかし $d \geq 1$ に対する cbind(..., deparse.level = d) 中のような引数名の逆構文解析からではなく).

メソッド

signature(x = "ANY", y = "ANY") R の内部コードを使う既定メソッド.

signature(x = "ANY", y = "missing") 一つの引数に対する R の内部コードを使う既定メソッド.

See Also

cbind, rbind; 更に **Matrix** パッケージ中の cBind, rBind.

Examples

```
cbind2(1:3, 4)
m <- matrix(3:8, 2,3, dimnames=list(c("a","b"), LETTERS[1:3]))
cbind2(1:2, m) # m からの dimnames を保つ

## rbind() と cbind() は今や rbind2()/cbind2() メソッドを使う
setClass("Num", contains="numeric")
setMethod("cbind2", c("Num", "missing"),
          function(x,y, ...) { cat("Num-miss--meth\n"); as.matrix(x)})
setMethod("cbind2", c("Num","ANY"), function(x,y, ...) {
    cat("Num-A.--method\n") ; cbind(getDataPart(x), y, ...) })
setMethod("cbind2", c("ANY","Num"), function(x,y, ...) {
    cat("A.-Num--method\n") ; cbind(x, getDataPart(y), ...) })

a <- new("Num", 1:3)
trace("cbind2")
cbind(a)
cbind(a, four=4, 7:9)# cbind2() を二回呼び出す

cbind(m,a, ch=c("D","E"), a*3)
cbind(1,a, m) # 警告付きで ok
untrace("cbind2")
```

Classes　　クラスの定義

Description

クラス定義は R オブジェクトのクラスの形式的定義を含むオブジェクトで，非形式的な S3 クラスと区別するために，普通 S4 クラスと参照される．このドキュメントは S4 クラスの概観を与える；クラス表現オブジェクトの詳細についてはクラス classRepresentation に対するヘルプを見よ．

メタデータ情報

クラスが定義されるとクラスに関する情報を格納するオブジェクトが保管される．クラスを定義するメタデータとして知られるこのオブジェクトはクラスの名前により保管されず (プログラマがその名前の生成関数を書くことを認めるために)，特別に構成された名前で保管される．クラス定義を吟味するためには getClass を呼び出す．メタデータ情報は以下を含む：

スロット： S4 クラスからのオブジェクト中に含まれるデータはクラス定義のスロットにより定義される．

オブジェクト中の各スロットはオブジェクトの成分である；リストの成分(つまり要素)の様にこれらは関数 slot() またはよりしばしば演算子 "@" で取り出したり設定できる．しかしながら，重要な点でそれらはリスト成分とは異なる．先ずスロットは名前でだけ参照でき位置では出来ず，リスト要素とは異なり部分マッチングは出来ない．

特定のクラスからの全てのオブジェクトは同じスロット名のセットを持つ；特にスロット名はクラス定義に含まれている．各オブジェクト中の各スロットは常に現在のクラスの定義中のこのスロットに対して指定されたクラスのオブジェクトである．単語 “is” は同名の R 関数(is) に対応し，スロット中のオブジェクトのクラスが定義で指定されたクラスまたは定義中のそれを拡張するあるクラス(サブクラス) と同じでなければならないことを意味する．

特別なスロット名 .Data はオブジェクトの ‘データ部’用である．データ部を持つクラスからのオブジェクトはクラスが R のオブジェクトタイプの一つか特別な擬似クラス matrix または array のどれかを含むことを指定することで定義される．普通クラスまたはそのスーパークラスの一つの定義はその contains 引数中にタイプや擬似クラスを含むからである．二番目の特殊スロット名 .xData は "environment" の様な特殊なタイプからの継承を可能にするために使われる．これらのクラスからのオブジェクトを持つ S3 メソッドの表現と挙動に関する詳細については非 S4 クラスからの継承に関する節を見よ．

ある種のスロット名は旧式の S3 オブジェクト中と明示的なクラスを持たない R オブジェクト中の属性に対応する，例えば names 属性．名前付きベクトルのサブクラスのような，属性が設定されるクラスを定義するならば， "names" をスロットとして含めるべきである．例えば "namedList" クラスの定義を見よ．そうした名前の設定に names() 付値を使うともし名前付きのスロットがなければ警告が出，問題のオブジェクトがベクトルタイプでなければエラーになる．"names" と呼ばれるスロットはどこでも使えるが，それがスロットとして付値された時だけで既定の names() 付値を使ってではない．

スーパークラス： クラスの定義はこのクラスを拡張するクラスである スーパークラスを含む．例えばクラス Fancy は，もし Fancy クラスからのオブジェクトが Simple クラスの全て(そして恐らくそれ以上)の機能を持てばクラス Simple を拡張する．特に，そして非常に有用に， Simple オブジェクトに対して動作する任意の定義されたオブジェクトは同様に Fancy オブジェクトにも適用できる．

この関係は同じことであるが Simple は Fancy のスーパークラスである，または Fancy は Simple のサブクラスであると表現できる．

あるクラスの直接のスーパークラスは明示的に定義されたスーパークラスである．直接のスーパークラスは三つの方法で定義できる．最も普通にはスーパークラスはサブクラスを作る setClass への呼び出し中の contains= 引数中に並べられる．この場合サブクラスはスーパークラスの全てのスロットを含み，クラス間の関係はそれが実際そうであるように 単純であると呼ばれる；スーパークラスは又 setIs への呼び出しで明示的に定義できる；この場合関係はサブクラスからスーパークラスへ向かうための指定されたメソッドを必要とする．三つ目として，クラスの合併は合併の全てのメンバーのスーパークラスになる．この場合も関係は単純であるが，関係はスーパークラスが定義された時に定義され， contains= 機構をもちいた時の様にサブクラスが作られた時では無いことを注意する．

スーパークラスの定義はまた潜在的にそれ自体の直接のスーパークラスを含む．これらはオリジナルのクラスから距離2のスーパークラスを考えら(そして表示され)れる；それらの直接のスーパークラスは距離3にある，云々．これら全てはメソッド選択等の目的に取っては正当なスーパークラスである．

contains= 引数へスーパークラスの名前を含めることでスーパークラスが定義される時，このクラスからのオブジェクトはそれ自体のクラスに対して定義された全てのスロットとそして その全てのスーパークラスに対して定義された全てのスロットを持つ．

クラスとその特定のスーパークラス間の関係の情報はクラス SClassExtension のオブジェクトとしてエンコードされる．スーパークラス(ある場合にはサブクラス)に対するそうしたオブジェクトのリストはクラスを定義するメタデータオブジェクト中に含まれる．もしこれらのオブジェクトを用いた計算が必要なら(例えば距離を比較する)，関数 extends を引数 fullInfo=TRUE で呼び出す．

プロトタイプ： new への呼び出しで作られたクラスからのオブジェクトはクラスに対するプロトタイプオブジェクトと new への呼び出し中の追加引数により定義される．引数は関数 initialize に対するそのクラスのメソッドに渡される．

オブジェクトを表現する各クラスはそのクラスに対するプロトタイプオブジェクトをを含む (仮想クラスに対してはプロトタイプオブジェクトは NULL かもしれない)．プロトタイプオブジェクトはそのクラスの全てのスロットに対して値を持たなければならない．既定では対応するスロットクラスのプロトタイプがある．しかしながらクラスの定義は任意のスロットに対する任意の適正なオブジェクトを指定できる．

仮想的クラス；基本クラス

new への呼び出しで実際のオブジェクトが何も作られない仮想的なクラスがあり，実際には非常に重要なプログラミングツールである．それらはある種のプログラミング挙動を共有したい通常のクラスを，挙動をどのように実装するかを制約する必要無しにまとめるために使われる．仮想クラスの定義は希望すればスロットを含むことが出来る (オブジェクトを完全に定義することなしにある種の共通の挙動を提供するために — 例えばクラス traceable を見よ)．

仮想クラスの単純で有用な形式は合併クラスであり， setClassUnion への呼び出しで一つ又はそれ以上のサブクラスを羅列することで定義される仮想クラスである (合併クラスを拡張するクラス)．合併クラスはサブクラスとして基本的なオブジェクトタイプを含むことが出来る(その定義はさもなければ封印される)．

幾つかの‘基本’クラスがあり， R 中で登場する通常の種類のデータに対応する．例えば "numeric" は数値ベクトルに対応するクラスである．他のベクトル基本クラスは "logical", "integer", "complex", "character", "raw", "list" そして "expression" である．ベクトルクラスに対するプロトタイプは対応するタイプの長さ0のベクトルである．基本クラスはプロトタイプオブジェクトがクラス自体からのものであるという点で普通ではない．

ベクトルクラスに加えて "function" や "call" のような言語中のオブジェクトに対応する基本クラスがまたある．これらのクラスは仮想的クラス "language" のサブクラスである．最後に "environment" と "externalptr" の様な “異常”なオブジェクトに対するオブジェクトタイプと対応する基本クラスがある．これらのオブジェクトは言語の機能的挙動に従わない；特にそれらはコピーされず，従って局所的に定義される属性やスロットを持つことが出来ない．

これら全てのクラスは任意の他のクラス定義に対するスロットやスーパークラスとして使うことができるが，それら自体は明示的なクラスを伴わない．異常なオブジェクトタイプに対しては，以下で述べる継承を可能にするために特別な機構が使われる．

非 S4 クラスからの継承

クラス定義は正規の S4 以外のクラスも拡張でき，普通それらを setClass への contains= 引数として指定する．そうしたクラスの三種類のグループは異なった挙動を持つ：

1. S3 クラス，これは事前に setOldClass の呼び出しで登録されている必要がある (これが行われているかどうかは getClass の呼び出しでチェックでき，これは oldClass を拡張するクラスを返すべきである)．
2. 典型的にはベクトルタイプの R のオブジェクトタイプの一つで，これはすると S4 オブジェクトのタイプを定義するが， S4 オブジェクトに対するタイプとしては直接使えない environment の様なタイプも定義する．下を見よ．
3. 擬似クラス matrix と array の一つで，任意のベクトルタイプに dim と dimnames 属性を加えたオブジェクトを意味する．

この節は S4 クラスと旧式の S3 クラスを用いた計算をそうしたクラスをスーパークラスとして用いて結合するアプローチを解説する．デザインの目標は S4 クラスが S3 メソッドと既定の計算を継承することを可能な限り一貫性のある形式で許すことである．

R 中の S4 と S3 コードをより一貫性のあるものにする一般的な努力の一部として，S3 総称的関数 (UseMethod を呼び出すもの)に対してかまたは S3 メソッドを選択適用するプリミティブ関数のどれかに対してか S4 クラスが非既定の S3 メソッドの第一引数として使われた時，そのメソッドに対して適正なオブジェクトを提供する努力が行われる．特に，もし S4 クラスが S3 クラスや matrix 又は array を拡張し，そしてこれらのクラスにマッチする S3 メソッドがあれば，S3 クラスの形式的定義が無いという条件の下で可能な範囲内で S4 オブジェクトは適正な S3 オブジェクトに強制変換される．

例えば "myFrame" が S3 クラス "data.frame" を setClass の contains= 引数中に含む S4 クラスであると仮定する．この S4 クラスからのオブジェクトが関数，例えば "data.frame" に対する S3 メソッドを持つ as.matrix に渡されると，UseMethod に対する内部コードはオブジェクトをデータフレームに変換する；特にそのクラス属性が S3 クラス(可能性として複数のクラス名を含む)に対応するベクトルである S3 オブジェクトに変換する．同様に "matrix" または "array" を継承する S4 オブジェクトに対しては，S4 オブジェクトは適正な S3 行列または配列に変換される．

S4 オブジェクトが既定の S3 メソッドに渡された時は変換は適用 されないことを注意しよう．ある種の S3 総称的関数は S4 オブジェクトを含む一般的なオブジェクトを処理しようと試みる．又選択された S3 メソッドに対応しない S4 オブジェクトには如何なる変換も行われない；特に S3 クラスか基本的タイプのどれかを含まないクラスからのオブジェクトに対してはそうである．変換の詳細については asS4 を見よ．

明示的な S3 総称的関数に加えて， S3 メソッドはプリミティブとして実装されている様々な演算子と関数に対して定義されている．これらのメソッドはある内部的な C コードにより，一部は実際の S3 総称的関数と同じコードを通じて，そして一部は特別な配慮(例えば二項演算子の二つの引数がメソッドを探す時に吟味される) により選択適用される．S4 オブジェクトを S3 メソッドに適合させる同じ機構が又これらの計算に適用されるが，適当な S3 クラスやタイプに拡張されない S4 オブジェクトが二項演算子に渡されるとエラーが発生する等の少数の例外がある．

この節の残りは基本的オブジェクトタイプからの継承に対する機構を解説する．行列と配列擬似クラスや時系列からの継承については matrix や array を見よ．対応する S3 クラスからの継承に関する詳細は setOldClass を見よ．

R の基本的データタイプの一つを直接かつ単純に含むクラスからのオブジェクトは対応するそのタイプの .Data スロットを暗黙のうちに持ち，他のスロットを変更せずにデータ部分を取り出したり置き換えることが許される．もしタイプが属性を受け入れ普通に複製できるのなら，継承はまたオブジェクトのタイプも決める；もしクラス定義が普通のタイプに対応する .Data スロットを持てば，スロットのクラスがオブジェクトのタイプを決定する (つまり typeof(x) の値)．そうしたクラスに対しては .Data は擬似スロットである；つまりそれを取り出したり設定するとデータ中の非スロットデータを変更する．関数 getDataPart と setDataPart はよりクリーンであるが，データ部分を処理する本質的に同値な方法である．

基本的タイプをこのように拡張することはオブジェクトが S4 メソッドと並んで対応するタイプに対する旧式のコードを使うことを可能にする． .Data に対しては任意の基本的タイプが使用可能であるが，幾つかのタイプは通常のオブジェクトのようには振るまわないので異なって扱われる；例えば "NULL"，環境，そして外部ポインターが該当する．

クラスはこれらのタイプをそれ自体内部的に定義された S4 クラスを継承するスロット .xData を持つことで拡張する．このスロットは実際には継承されたタイプのオブジェクトを含み，そのタイプの参照セマンティックスから計算を防御する．非標準的なオブジェクトタイプへの強制変換はすると他のタイプやクラスの "simple" な抱合ではなく実際の計算を必要とする．目的はプログラマーが機構を考慮する必要が無いが，任意のオブ

ジェクトタイプからの継承を検出するために S4 オブジェクトのタイプを明示的に使うべきでは無いということを含意する．代わりに is や類似の関数を使う．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョン．)

Chambers, John M. and Hastie, Trevor J. eds (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole (S3 クラスに対しては Appendix A．)

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (絶版．) (ベクトル，行列，配列，そして時系列オブジェクトの記述．)

See Also

メソッドに対する同様の議論については Methods，クラス定義の指定の詳細は setClass，is, as, new, slot

---

classesToAM クラス定義のスーパークラスに対する隣接行列を計算

Description

クラス名のベクトルかクラス定義のリストを与えて，関数はこれらのクラスのスーパークラスの隣接行列を返す；つまり，クラス名が行名と列名で，もし列 j 中のクラスが行 i 中のクラスの直接のスーパークラスならば 1 で，さもなければ 0 になる．

行列はクラス定義のスロット contains が含意する情報を持つが，更なる解析に対してしばしばより便利な形式でである；例えば隣接行列はパッケージや他のソフトウェア中で関係のグラフ表現を構成するのに使われる．

Usage

```
classesToAM(classes, includeSubclasses = FALSE,
       abbreviate = 2)
```

Arguments

classes	クラス名の文字列ベクトルかリストで，その要素はクラス名でもクラス定義でも良い．リストは，例えば，クラス名に対するパッケージスロットを含むために便利である．
includeSubclasses	論理フラグ；もし TRUE ならば，行列はスーパークラスと並んで指定されたクラスの全ての既知のサブクラスを含む．引数はまた classes と同じ長さの全てのクラスではなくあるサブクラスを含むための論理値ベクトルでも良い．
abbreviate	返される行列の行と/または列ラベルの省略形を制御する：値 0, 1, 2 または 3 は行も列も省略しない，行を省略，列を省略，そして行も列も省略する，ことを意味する．クラス名は一文字より長くなりがちであるため，既定の 2 は広がったプリントを作るために行列のプリントに対して有用である．値 0 か 3 はグラフを作るのに適している (3 はある種のグラフプロット用ソフトウェアは非常に小さいフォントサイズのラベルを作りやすいのを避ける).

Details

クラスの各々に対して，計算はクラス定義から全てのスーパークラス名を獲得し，これらのクラス定義中のエッジを見つける；つまり距離1の全てのスーパークラス．隣接行列の対応する要素は1に設定される．

個別のクラス定義に対する隣接行列は融合される．二つの可能な非一貫性に注意する．どちらも可能性として異なったパッケージからの同じ名前のクラスがある場合を除き問題を惹き起こすべきではない．エッジは各スーパークラス定義から計算されるので，情報は距離 > 1 の拡張要素からの可能な推測を上書きする (そしてそうあるべきである)．引数中の引き続く行列が融合された時は，計算は現在非一貫性をチェックしない —これは可能性のある同じ名前の重複クラスが混乱を招くかもしれない部位である．将来のバージョンは一貫性のチェックを含むかもしれない．

Value

解説されたように要素が 0 か 1 の行列で，非ゼロの値は列に対応するクラスが対応する行のクラスの直接のスーパークラスであることを指示する．行と列の名前はクラス名である (パッケージスロット無しの)．

See Also

背景にあるクラス定義からの情報については extends と classRepresentation.

Examples

```
## "standardGeneric" と "derivedDefaultMethod" の
## スーパークラスとサブクラス
am <- classesToAM(list(class(show), class(getMethod(show))), TRUE)
am

## Not run:
## 次の関数は Bioconductor パッケージ Rgraphviz に依存する
plotInheritance <- function(classes, subclasses = FALSE, ...) {
    if(!require("Rgraphviz", quietly=TRUE))
      stop("Only implemented if Rgraphviz is available")
    mm <- classesToAM(classes, subclasses)
    classes <- rownames(mm); rownames(mm) <- colnames(mm)
    graph <-  new("graphAM", mm, "directed", ...)
    plot(graph)
    cat("Key:\n", paste(abbreviate(classes), " = ", classes, ", ",
        sep = ""),  sep = "", fill = TRUE)
    invisible(graph)
}

## "graph" パッケージのクラス継承のプロット
require(graph)
plotInheritance(getClasses("package:graph"))


## End(Not run)
```

Classes_Details *Class Definitions*

Description

Class definitions are objects that contain the formal definition of a class of R objects, usually referred to as an S4 class, to distinguish them from the informal S3 classes. This document gives an overview of S4 classes; for details of the class representation objects, see help for the class `classRepresentation`.

Metadata Information

When a class is defined, an object is stored that contains the information about that class. The object, known as the *metadata* defining the class, is not stored under the name of the class (to allow programmers to write generating functions of that name), but under a specially constructed name. To examine the class definition, call `getClass`. The information in the metadata object includes:

Slots: The data contained in an object from an S4 class is defined by the *slots* in the class definition.

Each slot in an object is a component of the object; like components (that is, elements) of a list, these may be extracted and set, using the function `slot()` or more often the operator `"@"`. However, they differ from list components in important ways. First, slots can only be referred to by name, not by position, and there is no partial matching of names as with list elements.

All the objects from a particular class have the same set of slot names; specifically, the slot names that are contained in the class definition. Each slot in each object always is an object of the class specified for this slot in the definition of the current class. The word "is" corresponds to the R function of the same name (`is`), meaning that the class of the object in the slot must be the same as the class specified in the definition, or some class that extends the one in the definition (a *subclass*).

A special slot name, `.Data`, stands for the 'data part' of the object. An object from a class with a data part is defined by specifying that the class contains one of the R object types or one of the special pseudo-classes, `matrix` or `array`, usually because the definition of the class, or of one of its superclasses, has included the type or pseudo-class in its `contains` argument. A second special slot name, `.xData`, is used to enable inheritance from abnormal types such as `"environment"` See the section on inheriting from non-S4 classes for details on the representation and for the behavior of S3 methods with objects from these classes.

Some slot names correspond to attributes used in old-style S3 objects and in R objects without an explicit class, for example, the `names` attribute. If you define a class for which that attribute will be set, such as a subclass of named vectors, you should include `"names"` as a slot. See the definition of class `"namedList"` for an example. Using the `names()` assignment to set such names will generate a warning if there is no names slot and an error if the object in question is not a vector type. A slot called `"names"` can be used anywhere, but only if it is assigned as a slot, not via the default `names()` assignment.

Superclasses: The definition of a class includes the *superclasses* —the classes that this class extends. A class `Fancy`, say, extends a class `Simple` if an object from the `Fancy` class has all the capabilities of the `Simple` class (and probably some more as well). In particular, and very usefully, any method defined to work for a `Simple` object can be applied to a `Fancy` object as well.

This relationship is expressed equivalently by saying that `Simple` is a superclass of `Fancy`, or that `Fancy` is a subclass of `Simple`.

The direct superclasses of a class are those superclasses explicitly defined. Direct superclasses can be defined in three ways. Most commonly, the superclasses are listed in the `contains=` argument in the call to `setClass` that creates the subclass. In this case the subclass will contain all the slots of the superclass, and the relation between the class is called *simple*, as it in fact is. Superclasses can also be defined explicitly by a call to `setIs`; in this case, the relation requires methods to be specified to go from subclass to superclass. Thirdly, a class union is a superclass of all the members of the union. In this case too the relation is simple, but notice that the relation is defined when the superclass is created, not when the subclass is created as with the `contains=` mechanism.

The definition of a superclass will also potentially contain its own direct superclasses. These are considered (and shown) as superclasses at distance 2 from the original class; their direct superclasses are at distance 3, and so on. All these are legitimate superclasses for purposes such as method selection.

When superclasses are defined by including the names of superclasses in the `contains=` argument to `setClass`, an object from the class will have all the slots defined for its own class *and* all the slots defined for all its superclasses as well.

The information about the relation between a class and a particular superclass is encoded as an object of class `SClassExtension`. A list of such objects for the superclasses (and sometimes for the subclasses) is included in the metadata object defining the class. If you need to compute with these objects (for example, to compare the distances), call the function `extends` with argument `fullInfo=TRUE`.

Prototype: The objects from a class created by a call to `new` are defined by the *prototype* object for the class and by additional arguments in the call to `new`, which are passed to a method for that class for the function `initialize`.

Each class representation object contains a prototype object for the class (although for a virtual class the prototype may be `NULL`). The prototype object must have values for all the slots of the class. By default, these are the prototypes of the corresponding slot classes. However, the definition of the class can specify any valid object for any of the slots.

Basic classes

There are a number of 'basic' classes, corresponding to the ordinary kinds of data occurring in R. For example, `"numeric"` is a class corresponding to numeric vectors. The other vector basic classes are `"logical"`, `"integer"`, `"complex"`, `"character"`, `"raw"`, `"list"` and `"expression"`. The prototypes for the vector classes are vectors of length 0 of the corresponding type. Notice that basic classes are unusual in that the prototype object is from the class itself.

In addition to the vector classes there are also basic classes corresponding to objects in the language, such as `"function"` and `"call"`. These classes are subclasses of the virtual class `"language"`. Finally, there are object types and corresponding basic classes for "abnormal" objects, such as `"environment"` and `"externalptr"`. These objects do not follow the functional behavior of the language; in particular, they are not copied and so cannot have attributes or slots defined locally.

All these classes can be used as slots or as superclasses for any other class definitions, although they do not themselves come with an explicit class. For the abnormal object types, a special mechanism is used to enable inheritance as described below.

Inheriting from non-S4 Classes

A class definition can extend classes other than regular S4 classes, usually by specifying them in the `contains=` argument to `setClass`. Three groups of such classes behave distinctly:

1. S3 classes, which must have been registered by a previous call to `setOldClass` (you can check that this has been done by calling `getClass`, which should return a class that extends

oldClass);

2. One of the R object types, typically a vector type, which then defines the type of the S4 objects, but also a type such as `environment` that can not be used directly as a type for an S4 object. See below.
3. One of the pseudo-classes `matrix` and `array`, implying objects with arbitrary vector types plus the `dim` and `dimnames` attributes.

This section describes the approach to combining S4 computations with older S3 computations by using such classes as superclasses. The design goal is to allow the S4 class to inherit S3 methods and default computations in as consistent a form as possible.

As part of a general effort to make the S4 and S3 code in R more consistent, when objects from an S4 class are used as the first argument to a non-default S3 method, either for an S3 generic function (one that calls `UseMethod`) or for one of the primitive functions that dispatches S3 methods, an effort is made to provide a valid object for that method. In particular, if the S4 class extends an S3 class or `matrix` or `array`, and there is an S3 method matching one of these classes, the S4 object will be coerced to a valid S3 object, to the extent that is possible given that there is no formal definition of an S3 class.

For example, suppose `"myFrame"` is an S4 class that includes the S3 class `"data.frame"` in the `contains=` argument to `setClass`. If an object from this S4 class is passed to a function, say `as.matrix`, that has an S3 method for `"data.frame"`, the internal code for `UseMethod` will convert the object to a data frame; in particular, to an S3 object whose class attribute will be the vector corresponding to the S3 class (possibly containing multiple class names). Similarly for an S4 object inheriting from `"matrix"` or `"array"`, the S4 object will be converted to a valid S3 matrix or array.

Note that the conversion is *not* applied when an S4 object is passed to the default S3 method. Some S3 generics attempt to deal with general objects, including S4 objects. Also, no transformation is applied to S4 objects that do not correspond to a selected S3 method; in particular, to objects from a class that does not contain either an S3 class or one of the basic types. See `asS4` for the transformation details.

In addition to explicit S3 generic functions, S3 methods are defined for a variety of operators and functions implemented as primitives. These methods are dispatched by some internal C code that operates partly through the same code as real S3 generic functions and partly via special considerations (for example, both arguments to a binary operator are examined when looking for methods). The same mechanism for adapting S4 objects to S3 methods has been applied to these computations as well, with a few exceptions such as generating an error if an S4 object that does not extend an appropriate S3 class or type is passed to a binary operator.

The remainder of this section discusses the mechanisms for inheriting from basic object types. See `matrix` or `array` for inhering from the matrix and array pseudo-classes, or from time-series. For the corresponding details for inheritance from S3 classes, see `setOldClass`.

An object from a class that directly and simply contains one of the basic object types in R, has implicitly a corresponding `.Data` slot of that type, allowing computations to extract or replace the data part while leaving other slots unchanged. If the type is one that can accept attributes and is duplicated normally, the inheritance also determines the type of the object; if the class definition has a `.Data` slot corresponding to a normal type, the class of the slot determines the type of the object (that is, the value of `typeof(x)`). For such classes, `.Data` is a pseudo-slot; that is, extracting or setting it modifies the non-slot data in the object. The functions `getDataPart` and `setDataPart` are a cleaner, but essentially equivalent way to deal with the data part.

Extending a basic type this way allows objects to use old-style code for the corresponding type as well as S4 methods. Any basic type can be used for `.Data`, but a few types are treated differently because they do not behave like ordinary objects; for example, `"NULL"`, environments, and external pointers. Classes extend these types by having a slot, `.xData`, itself inherited from an internally

defined S4 class. This slot actually contains an object of the inherited type, to protect computations from the reference semantics of the type. Coercing to the nonstandard object type then requires an actual computation, rather than the "simple" inclusion for other types and classes. The intent is that programmers will not need to take account of the mechanism, but one implication is that you should *not* explicitly use the type of an S4 object to detect inheritance from an arbitrary object type. Use is and similar functions instead.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

Methods_Details for analogous discussion of methods, setClass for details of specifying class definitions, is, as, new, slot

---

className 対応するパッケージを含むクラス名

Description

関数 className() はクラスの適正な参照を生成し，クラス定義を含むパッケージの名前を含む．クラス "className" からの返されたオブジェクトは，例えば setMethod を呼び出す時に，クラスの多重定義が存在する場合にクラスを参照する曖昧さの無い方法である．

関数 "multipleClasses" は異なったパッケージからの同じ名前を持つクラスの多重定義に関する情報を返す．

Usage

```
className(class, package)

multipleClasses(details = FALSE)
```

Arguments

class, package	クラスとそしてオプションでそれが所属するパッケージの名前の文字列．もし package が欠損し class 引数がパッケージスロットを持てば，それが使われる(特にこの場合にはクラス "className" からのオブジェクトを渡すとそれ自体を返すが，もし二番目の引数が提供されるとパッケージスロットが変更される)． もしパッケージ引数やスロットがなければ，クラスに対する定義が存在しなければならずパッケージを定義するのに使われる．もし多重定義があれば，一つが選ばれ他の可能性を与える警告がプリントされる．
details	もし既定の FALSE ならば， multipleClasses() は多重定義を持つ現在知られているクラスへの文字列ベクトルを返す． もし TRUE ならば，これらのクラス定義の名前付きリストが返される．リストの各要素はそれ自体対応するクラス定義のリストであり，リスト名としてパッケージ名を持つ．同一のクラス定義は“多重”定義とは考えられないことを注意する (下の詳細の議論を見よ)．

Details

内部的に使われるクラス定義のテーブルは同じ名前を持つが異なったパッケージに由来する多重定義を管理できる．もし同一のクラス定義が現れると，唯一つのクラス定義が保持される；これは setOldClass の呼び出し中で指定されている S3 クラスに関して最もしばしば起きる．真のクラス対しては，もし二つのパッケージが偶々独立に同じ名前を使っていると一般に多重クラス定義が不可避になる．

同じ名前を持つ別のパッケージ中のクラス定義を勝手に書き換えることは普通悪いアイデアである． R は二つの定義を保持し使う (バージョン 2.14.0 の様に)が，曖昧さは常に可能である．既存クラスを拡張子するが異なった名前を持つ新しいクラスを定義するのがより賢明である．

Value

className() への呼び出しはクラス "className" のオブジェクトを返す．

multipleClasses() への呼び出しはクラス定義の文字列ベクトルか名前付きリストを返す．どちらのケースも 0 より大きい返り値の長さの検査することは多重に定義されたクラスの存在のチェックである．

クラスからのオブジェクト

クラス "className" は "character" を拡張しスロット "package" を持ち，またクラス "character" である．

Examples

```
## Not run:
className("vector") # パッケージ "methods" から見つかる
className("vector", "magic") # OK, クラスは存在しないものの


className("An unknown class") # エラーを起こす

## End(Not run)
```

---

classRepresentation-class
クラスオブジェクト

Description

これらはオブジェクトのクラスの定義を保持するオブジェクトである．これらは関数 setClass の呼び出しによりメタデータとして構成され保存される．恐らく個別のスロットを眺める以外にはそれらを直接操作してはならない．

Details

クラス定義はメタデータとして様々なパッケージに保存される．追加のメタデータは継承に関する情報を提供する (setIs への呼び出しの結果)．クラス定義自体により含意される継承情報 (クラスは一つまたはそれ以上のクラスを含むので) は同様に自動的に構成される．

あるクラスが R セッション中で使われると，この情報はクラス定義を完成させるために集められる．完成したものははクラス "classRepresentation" の二番目のオブジェクトであり，セッションに対してまたは情報を変更する何かが起きるまでキャッシュされる．getClass への呼び出しはクラスの完成された定義を返す；getClassDef の呼び出しは保管された定義を返す(未完成の)．

特に完成されたものはクラスに対する上向きと下向きを指し示す継承情報をスロット contains と subclasses にそれぞれ描き込む．これは原則としてこれらは追加のサブクラスとスーパークラスを定義するかもしれないので，この情報はインストールされたパッケージに依存する可能性があることを注意することが重要である．

スロット

slots: このクラス中のスロットの名前付きリスト；リストの要素はスロットが所属する(または拡張する)クラスであり，リストの名前は対応するスロット名を与える．

contains: このクラスが'含む'クラスの名前付きリスト；リストの要素は SClassExtension のオブジェクトである．リストは直接の拡張か全ての現在知られている拡張かもしれない(詳細を見よ)．

virtual: 論理フラグ，もしこれが仮想クラスならば TRUE に設定される．

prototype: このクラスに対する標準的なプロトタイプを表すオブジェクト；つまりこのクラスに対する特別な引数無しの new の呼び出しにより返されるデータとスロット．プロトタイプオブジェクトを直接にいじくらないこと．

validity: オプションの，このクラスからのオブジェクトの妥当性を検査するのに使われる関数． validObject を見よ．

access: 制御情報にアクセスする．現在使われない．

className: クラスの名前の文字列．

package: クラスが属するパッケージ名の文字列．ほとんど常にクラスに対するメタデータが保管されているパッケージであるが，継承情報構成のような操作では内部的なパッケージ名規則．

subclasses: このクラスを拡張することが知られているクラスの名前付きリスト；リストの要素はクラス SClassExtension のオブジェクトである．リストは現在クラス定義を完成する時だけ埋められる (詳細を見よ)．

versionKey: クラス "externalptr" のオブジェクト；いつかは恐らくある種のバージョン情報を含むであろうが，現在は使われない．

sealed: クラス "logical" のオブジェクト；このクラスは封印されているか? もしそうならどんな修正も許されない．

See Also

クラス定義中の情報を提供するには関数 setClass を見よ．クラス情報のより基本的な議論には Classes.

Documentation	クラスとメソッドに対するオンラインドキュメントを使う・作る

Description

メソッドパッケージのソフトウェアで作られたクラスとメソッドを記述する特別なドキュメントを提供することが出来る．このドキュメントにアクセスしそれを R のヘルプファイル中に作るテクニックがここで解説される．

クラスとメソッドに関するドキュメントを得る

クラス定義，総称的関数の特定のメソッド，そして総称的関数の一般的な議論に対するオンラインヘルプを問い合わせることが出来る．これらの要求は ? 演算子を使う (演算子の一般的な記述に関しては `help` を見よ)．勿論対応するトピックに関する何らかのドキュメントが あるかどうかは実装者次第である．

クラスに関するドキュメントは ? の左に引数 `class` を，右にクラス名を使う；例えば，

```
class ? genericFunction
```

はクラス `"genericFunction"` に関するドキュメントを問い合わせる．

特定の関数に対して定義されたメソッドに対するドキュメントが必要なら，メソッドの一般的解説や特定のメソッド (つまり，実際の引数の特定のセットに対して選択される特定のメソッドのドキュメント)を問い合わせることが出来る．

全体的なメソッドのドキュメントは ? 演算子を `methods` を左側引数にそして関数名を右側に於いて呼び出すことで要求できる．例えば，

```
methods ? initialize
```

は `initialize` 関数に対するメソッドに対するドキュメントを問い合わせる．

特定のメソッドに関するドキュメントの問い合わせは `"?"` 演算子の右側引数に関数呼び出し表現式を与えることでなされる．引数に対するクラス名か，実際の呼び出し中で使うことを意図している表現式を与えるかどちらを好むかに応じて二つの形式がある．

例えば関数呼び出し `myFun(x, sqrt(wt))` を評価することを計画しておりこの呼び出しに対して使われるメソッドに関する何かを探したければ， `"?"` 演算子の右側に呼び出しを置く：

```
?myFun(x, sqrt(wt))
```

呼び出し自体に対してそうあるようにメソッドが選択され，そしてそのメソッドに対するドキュメントが要求される．もし `myFun` が総称的関数でなければ，関数に対する通常のドキュメントが要求される．

もしメソッドのドキュメントが欲しい実際のクラスを知っていれば，これらを引数表現の代わりに明示的に与えることが出来る．上の例では，もしクラス `"maybeNumber"` を持つ第一引数と二番目の `"logical"` に対するメソッドのドキュメントを必要とすれば `"?"` 演算子を今回は左側引数 `method` とクラス名を引数として右側に関数呼び出しとして用いる：

```
method ? myFun("maybeNumber", "logical")
```

再びメソッドが選ばれるが，今回は指定されたクラスに対応したメソッドのドキュメントが要求される．このバージョンは総称的関数に対してだけ動作する．

この二つの形式はそれぞれ長所を持つ．実際の引数を用いたバージョンは引数のクラスを特定(または推測)する必要がない．一方で，引数の評価は例に応じて少し時間がかかるかもしれない．クラス名を用いたバージョンはクラスを選ぶ必要があるが，しかし曖昧さが無い．これは提供されたクラスが仮想的クラスでも良いという点でより微妙な長所を持ち，その場合どの引数もこのクラスを明確には持たない．例えばクラス `"maybeNumber"` は合併クラスであっても良い (`setClassUnion` に対する例を見よ)．

どちらの形式でも，メソッドが継承の使用とグループ総称的関数を含み実際の計算でそうなるように選択される．詳細は適当なメソッドを見つけるのに使われる関数である `selectMethod` を見よ．

メソッドに対するドキュメントを書く

メソッドとクラスに対するオンラインドキュメントは上で説明されたクラスとメソッドのドキュメントに対する要求を実装するために R のドキュメント書式へのある種の拡張を使う．利用可能なマークアップ命令についてはドキュメント *Writing R Extensions* を見よ (もし自分のソフトウェアのドキュメント化の段階にいるのならばこのドキュメントをすでに参考にしているべきである).

解説される特定のマークアップ命令に加えて，特定の総称的関数に対して定義されたメソッドに対してドキュメントの骨格を持つ全体的な初期的ファイルを作ることができる：

```
promptMethods("myFun")
```

は myFun 関数に対して定義されたメソッドに対する骨格ドキュメントを持つファイル 'myFun-methods.Rd' を生成する． promptMethods からの出力はその関数に対する全てか殆どのメソッドを，総称的関数自体のドキュメントとは別に，一つのファイルに記述するのに適している．ファイルが埋められ自分のソースパッケージの 'man' 副ディレクトリに移動されると，メソッドのドキュメントに対する要求は，上で解説されたような特定のメソッドドキュメントに対しても

```
methods ? myFun
```

で要求された全般的なドキュメントに対してもそのファイルを使う． promptMethods を使う必要は無いが，もし使えば作られた全てのファイルを使いたくないかもしれない：

- もし総称的関数自体に対するドキュメントを含むファイル中にメソッドのドキュメントを入れたければ，カットアンドペーストで \alias 行と既存のファイルに対して promptMethods により作られたファイルからの Methods 節を移動することが出来る.
- 一方で，もしこれらが補助的なメソッドで，付け加えられたか修正されたソフトウェアだけをドキュメント化したければ，関心のあるメソッドに対するもの以外の全ての \alias 行と， Methods 節中の対応する \item を除く全ての項目を取り除くべきである．この場合普通最初の \alias 行も同様に取り除く．なぜならこれはこの関数に対する一般的なメソッドのドキュメントに対するマーカーだからである (例では '\alias{myfun-methods}').

一つ又は複数のメソッドに対するドキュメントをもし特定の R ドキュメントファイルに差し向けたければ，適当なエイリアスを挿入する.

---

dotsMethods　　メソッドシグネチャ中の *list("...")* の使用

Description

R 関数中の "..." 引数はゼロ，一つまたはそれ以上の実際の引数(つまり，オブジェクト)にマッチするという意味で特別に扱われる， "..." を総称的関数のシグネチャとして認める機構が R に加えられてきた．そうした関数に対して定義されたメソッドは "..." にマッチする全ての引数が指定されたクラス由来かそのクラスのサブクラス由来である時選択され呼び出される.

シグネチャ中の "..." の使用

R のバージョン 2.8.0 以来，S4 メソッドが特殊引数 "..." に対応する選択適用出来るようになった．現在 "..." は他の形式的引数と混用は出来ない：総称的関数のシグネチャは "..." だけか，または "..." を含まないかである． (この制限は将来除かれるかもしれない．)

適当な総称的関数をあたえた時，メソッドが通常のように setMethod の呼び出しで指定される．メソッド定義は "..." に対応する全ての引数がメソッドシグネチャ中かまたはそのクラスを拡張するクラスからのものと仮定して書かれるべきである (つまり，そのクラスのサブクラス).

典型的にはメソッドは "..." を別の関数に引き渡すかまたは引数のリストを作りその上で繰り返す．下の例を見よ．

一つより多い既存のクラスに対して適当な計算があるならば，便利なアプローチは setClassUnion の呼び出しを用いてこれらのクラスの合併クラスを定義することである．下の例を見よ．

メソッド選択と "..." に対する選択適用

一般的な議論は Methods を見よ．以下はそのドキュメントの "メソッド選択と提供"節を読んでいることを前提としている．

"..." に関するメソッド選択は setMethod の呼び出し中の単一のクラスで指定される．もし "..." の全ての実際の引数がこのクラスを持てば，対応するメソッドは直接選択される．

さもなければ，各引数のクラスとそのクラスのスーパークラスが "..." 引数の最初から始めて計算される．最初の引数に対しては適格なメソッドは任意のクラスに対するものである．引き続くそれまでに考慮されていないクラスを導入する各々の引数に対しては適格なメソッドは更に引数のクラスかスーパークラスにマッチするものに制限される．もし適格なクラスがそれ以上存在しなければ繰り返しは中断され，もし存在すれば，既定メソッドが選択される．

繰り返しの最後で，一つまたはそれ以上のメソッドが適合するかもしれない．もし複数あれば，選択は実際の引数に最も距離が近いメソッドを探す．任意の継承メソッドはある距離に対応し，クラス定義の contains スロットから利用できる．同じクラスが一つ以上の引数に対して起こり得るので，それに伴う複数の距離が存在し得る．それらを結びつけるのは任意であることが避けられない：現在の計算は最少の距離を使う．従って，例えばもしあるメソッドが一つの引数に直接に，一つが最初の世代のスーパークラスにそして別のが二代目のスーパークラスにマッチすれば，距離は 0, 1 そして 2 である．現在の選択計算はこのメソッドに対しては距離 0 を使う．特にこの選択基準は引数クラスの一つまたはそれ以上と正確にマッチするメソッドを使う傾向がある．

通常のメソッド選択を使うのと同様に，同じ距離を持つ複数のメソッドが存在し得る．警告メッセージが出されメソッドの一つが選ばれる (最初に出会ったもの，これはこの場合かなり任意である).

計算は全ての引数を吟味するが，選択適用の本質的なコストは引数中の異なったクラス数と共に上昇するが，引数の数が大きい時には引数の数より恐らくかなり小さくなることを注意する．

実装の詳細

"..." に関するメソッド選択適用は R のバージョン 2.8.0 で導入された．対応する選択と適用の最初の実装は新しい機構を研究中の間の融通性のため R 関数を用いてであった．この実装では setGeneric のローカルバージョンが総称的関数の環境中に挿入される．ローカルバージョンはメソッドを上の基準に従い選択され，そのメソッドを総称的関数の

環境から呼び出す．これは "..." が含まれていない時は C による実装により取られる挙動と少し異なる．必要な余分の計算時間を別として， C バージョンによって構成される特別な文脈 (これは R コードでは正確には復元できない)とは異なり，メソッドは真の関数呼び出し中で評価される．しかしながら，異なった計算結果が得られたような状況はこれまで遭遇していないし，非常にありそうもない．

"..." 以外の引数に対するメソッド選択適用は全てのメソッドのテーブル中で継承メソッドをソートすることで キャッシュされ，そこではそれは実引数中のクラスの組み合わせを持つ次の選択に際して見出される． (しかし継承探索に対しては使われない)． "..." に基づくメソッドもキャッシュされるが，しかし全くすぐには見つからない．注意されたように，選ばれたメソッドは "..." 引数中に登場するクラスのセットにだけ依存する．これらのクラスの各々は一度もしくはそれ以上現れることが出来，実際の引数のクラスの多くの組み合わせが同じ実効的なシグネチャに対して登場する．選択の計算は先ず遭遇した異なったクラスを計算しソートする．これはメソッドのテーブル中にキャッシュされるラベルを与え，継承クラスの最初の生起後の更なる探索を避ける． showMethods への呼び出しはそうした継承メソッドを展示する．

意図するところは "..." 特性はそれらに対する十分な経験が得られた時に標準的 C コードに付け加えられるということである．同時に "..." とシグネチャ中の他の引数のコンビネーションがサポートされるかもしれない．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

メソッドの一般的議論は Methods とそこからのリンクを見よ．

Examples

```
cc <- function(...)c(...)

setGeneric("cc")

setMethod("cc", "character", function(...)paste(...))

setClassUnion("Number", c("numeric", "complex"))

setMethod("cc", "Number", function(...) sum(...))

setClass("cdate", contains = "character", representation(date = "Date"))

setClass("vdate", contains = "vector", representation(date = "Date"))

cd1 <- new("cdate", "abcdef", date = Sys.Date())

cd2 <- new("vdate", "abcdef", date = Sys.Date())

stopifnot(identical(cc(letters, character(), cd1),
          paste(letters, character(), cd1))) # "character" メソッド

stopifnot(identical(cc(letters, character(), cd2),
```

```
                    c(letters, character(), cd2)))
# 既定, "vdate" は "character" を拡張しないから

stopifnot(identical(cc(1:10, 1+1i), sum(1:10, 1+1i))) # "Number" メソッド

stopifnot(identical(cc(1:10, 1+1i, TRUE), c(1:10, 1+1i, TRUE))) # 既定

stopifnot(identical(cc(), c())) # どの引数も既定メソッドを意味しない

setGeneric("numMax", function(...)standardGeneric("numMax"))

setMethod("numMax", "numeric", function(...)max(...))
# 複素数値データには使えない
setMethod("numMax", "Number", function(...) paste(...))
# 複素引数内では選択されるべきではない

stopifnot(identical(numMax(1:10, pi, 1+1i), paste(1:10, pi, 1+1i)))
stopifnot(identical(numMax(1:10, pi, 1), max(1:10, pi, 1)))

try(numMax(1:10, pi, TRUE)) # エラーになるべき：既定メソッドが無い

## paste() の総称的バージョン, "..." 引数に対して選択適用：
setGeneric("paste", signature = "...")

setMethod("paste", "Number", function(..., sep, collapse) c(...))

stopifnot(identical(paste(1:10, pi, 1), c(1:10, pi, 1)))
```

---

environment-class　　クラス *list("environment")*

Description

R 環境に対する形式的クラス.

クラスからのオブジェクト

オブジェクトは形式 new("environment", ...) の呼び出しで作ることが出来る. ...中の引数はもしあれば名前付きでなければならず新しく作られた環境に付値される.

メソッド

coerce signature(from = "ANY", to = "environment"): as.environment を呼び出す.

initialize signature(object = "environment"): 新しい環境中の付値を実装. object 引数は無視されることを注意する；環境はコピーに対して保護されていないため, 新しい環境が 常に作られる.

See Also

new.env

envRefClass-class　　クラス *list("ënvRefClass")*

Description

参照セマンティクスを用いて R オブジェクトを実装するクラスをサポートする

注意：

ここで解説されるソフトウェアは初期バージョンである．最終ゴールは R 自体か内部システムインタフェイスを使い参照スタイルのクラスをサポートすることである．現在の実装 (R バージョン 2.12.0)は予備的であり変更の可能性があり，現在 R だけの実装のみを含む．開発者はこのソフトウェアを使って実験することを推奨されるが，ここでの記述は普通以上に変更の可能性がある．

クラスの目的

このクラスは R オブジェクトに対する基本的参照スタイルを実装する．オブジェクトは通常直接的にはこのクラスには由来せず， `setRefClass` の呼び出しにより定義されるサブクラスに由来する．下のドキュメントは実装を解説する技術的背景であるが，アプリケーションは `setRefClass` の下で解説されたインタフェイス，特にここで解説された `$` 演算子と欄へのアクセス関数，を使うべきである．

基本的な参照クラス

R に対する参照クラスのデザインはこれらのクラスを参照，欄，そしてクラスメソッドの実装に対する機構に従って分割する．この機構の各バージョンは 基本参照クラスにより定義されており，これはメソッドのセットと `setRefClass` により使われるある更なる情報を提供しなければならない．

必要とされるメソッドはオブジェクト中の欄を取得し設定する演算子 `$` と `$<-`，そしてオブジェクトを初期化する `initialize` に対するものである．

これらのメソッドをサポートするために，基本的な参照クラスはデータをオブジェクト中の欄に保管し回収するある実装機構を持つ必要がある．この機構は参照のセマンティクスと一貫性がある必要がある；つまり，あるオブジェクトの中身に行われた変更は大局的で，変更が行われた関数呼び出しに対して局所的ではなく，そのオブジェクトにアクセスする全てのコードから見えなければならない．下で説明されるように，クラス `envRefClass` は environment オブジェクトの特殊な使用を用いて参照セマンティクスを実装する．他の基本的な参照クラスはクラスに対する参照セマンティクスを用いて Java や C++ の様な言語へのインタフェイスを使うかもしれない．

普通 R ユーザは `$` 演算子を使いクラスにクラスメソッドを起動できる． `$` に対する基本的な参照クラスはこれが可能になるようにする必要がある．本質的に，演算子はオブジェクトとクラスメソッド名に対応する R 関数を返さなければならない．

クラスメソッドは欄が “get” と “set” メソッドでアクセスできるという意味でデータ抽象化の実装を含むかもしれない．基本的な参照クラスはこの機能を引数一つの関数に対するその定義中のスロット `"fieldAccessorGenerator"` を設定することで提供する．この関数は欄名のベクトルを引数とすることで `setRefClass` により呼び出される．生成関数は定義されたアクセス関数のリストを返さなければならない．取得演算に対応する要素は引数無しで起動され対応する欄を取り出すべきである；設定演算は欄に付値される値である引数一つで起動される．実装はそれがメソッド起動の引数ではないためオブジェクトを提供する必要がある．現在 `envRefClass` が使う機構は下で解説される．

サポートクラス

二つの仮想的クラスが参照クラスのテストのために提供される： `is(x, "refClass")` は x がここで解説された参照クラス機構を使い定義されたクラスに由来するかどうかをテストする； `is(x, "refObject")` はオブジェクトが，先のクラスとまた `"environment"` の様な参照セマンティクスを持つ R タイプを継承するクラスを含み，一般的参照セマンティクスを持つかどうかをテストする．

インストールされるクラスメソッドは `"classMethodDefinition"` オブジェクトで，クラスメソッドとしての関数名とこのメソッドから呼び出される他のクラスメソッドを特定する．後者の情報はクラスが `codetools` 推奨パッケージを使うことで定義された時にヒューリスティックに決定される．このパッケージは参照クラスが定義される時にインストールされるべきであるが，既存の参照クラスを使うためには不要である．

Author(s)

John Chambers

`evalSource` パッケージを再インストールせずにソースファイルからの関数定義を使う

Description

ソースファイルからの関数と/またはメソッドの定義を `trace` 機構を使いパッケージに挿入される．典型的には，これは大きなパッケージを再インストールすること無しに幾つかの関数の修正バージョンのテストやデバッグを許す．

Usage

```
evalSource(source, package = "", lock = TRUE, cache = FALSE)

insertSource(source, package = "", functions = , methods = ,
           force = )
```

Arguments

`source`	新しい関数とメソッド定義を見つけるために `evalSource` により構文解析と評価がされるファイル． `insertSource` への引数は `evalSource` への先の呼び出しが返すクラス `"sourceEnvironment"` のオブジェクトであって良い．詳細はクラスに関する節を見よ．
`package`	オプションで，新しいコードが対応しその中にそれが挿入されるパッケージの名前．もし欠損していればパッケージを推測することが試みられるが，それを提供するのが安全なアプローチである．検索パスに付加されていないパッケージの場合は，パッケージ名を提供する必要がある．
`functions, methods`	オプションで，挿入に使われる関数名の文字列． `functions` 引数に提供された名前はソース中に関数として定義されていると期待されている． `methods` 引数中に提供された名前に対して，メソッドのテーブル

が期待されている (setMethod の呼び出しにより作られるような，詳細節を見よ)；このテーブルからのメソッドは insertSource により挿入される．どちらのケースでも，改定された関数やメソッドはロードされた対応するパッケージ中のバージョンから異なる時だけ挿入される．

もし what が省略されると，ソースファイルファイルの評価の結果はパッケージの中身と比較される(詳細節を見よ).

lock, cache
: evalSource により取られる行動を制御するオプションの引数．もし lock が TRUE ならば，返されるオブジェクト中の環境はロックされ，従って全てのその拘束もロックされる．もし cache が FALSE ならば，メソッドとクラス定義の通常のキャッシュは source ファイルの評価中は抑制される．

既定の設定が一般に推奨され，ソースファイルのスナップショットとしてのオブジェクトの信憑性をサポートするために lock し，そして次に後でトレース機構を用い insertSource によりメソッド定義が挿入出来るようにする．

force
: もし FALSE なら，現在環境中にある関数だけが trace を用い再定義される．もし TRUE なら，他のオブジェクト/関数は単純に付値される．既定では，もし functions も methods 引数どちらも提供されなければ TRUE.

Details

source ファイルは構文解析され評価されるが，既定ではメソッドとその中に含まれるクラス定義の実際のキャッシュは抑制されるため，関数とメソッドは逆転できるような仕方でテストできる．もしすべてが上手くいけば，結果は付値されたオブジェクトとソースファイル中のメソッドとクラス定義に対応するメタデータを含む環境である．

この環境から，もしそれが持てばその名前空間中に，現在のセッション中か untrace の呼び出しでオリジナルのバージョンに戻すまで使われるために，オブジェクトがパッケージに挿入される．挿入は回復を可能にするため trace の内部バージョンの呼び出しで行われる．

トレース機構が使われるため，関数タイプのオブジェクト，関数自体か S4 メソッド，だけが挿入される．

functions と methods 引数が共に省略された時は， insertSource は source ファイルの評価結果から適当なオブジェクトを選択する．

全てのケースで，パッケージ中の対応するオブジェクトと異なるソースファイル中のオブジェクトだけが挿入される．“differ” の定義は引数リスト (既定の表現式を含む)か関数本体が同一ではないものである．メソッドのケースでは，パッケージ中の対応するシグネチャに対する特定のメソッドを持たない必要がある：比較はそのシグネチャに対して選択されるメソッドに対して行われる．

計算は提供されるソースファイルがオリジナルのパッケージソース中と同じであることを何も必要がないが，それがパッケージを改訂する時だけありそうでかつ意味を持つ．計算ではソースファイルを比較しない： source の評価で生成されたオブジェクトはオブジェクトとしてパッケージの中身と比較される．

Value

クラス "sourceEnvironment" のオブジェクトで， "environment" のサブクラス(クラスに関する節を見よ)．環境はソースファイルの評価の結果である 全てのオブジェクトを含む．クラスは又生成時間，ソースファイルそしてパッケージ名に対するスロットを持つ．将来の拡張はこれらのオブジェクトをバージョン管理や他のコードツールに対して使うかもしれない．

返されるオブジェクトはデバッグ中(そのトピックに関する節を見よ)か将来の insertSource への将来の呼び出し中の source 引数として使うことが出来る．もしある改訂された関数だけが最初の呼び出しで挿入されていると，他は環境とオプションで適当な functions そして/または methods 引数により，ソースファイルを再評価すること無く後の呼び出し中で挿入できる．

デバッグ

一旦 insertSource により関数かメソッドがパッケージ中に挿入されてしまうと，それは標準のデバッグツールで吟味できる；例えば debug か trace の様々なバージョン．

trace の呼び出しは追加の引数 edit = env を取るべきで，ここで env は evalSource の呼び出しにより返される値である．トレース機構がソースファイルからの改訂バージョンのインストールに使われており，引数を提供するとトレースされるのはオリジナルではなくこのバージョンであることが確証される．下の例を見よ．

トーレースをオフにするがソースバージョンを維持するには，例の中のように trace(x, edit = env) を使う．パッケージからのオリジナルバージョンに戻るには untrace(x) を使う．

クラス "sourceEnvironment"

このクラスからのオブジェクトは，関数と evalSource から生成されたメソッドのバージョンを取り出す環境として扱うことが出来る．オブジェクトはまた以下のスロットを持つ：

packageName: ソースコードが対応するパッケージの名前の文字列．

dateCreated: ソースファイルが評価された日時 (普通呼び出し Sys.time から)．

sourceFile: 使われたソースファイルの名前の文字列．

環境の使用は dateCreated を変更しないことを注意する．

See Also

基礎にある機構については trace，そして又少し似た目的に対して使われる edit= 引数；伝統的なデバッグスタイルを志向するテクニックに対してはこの関数と又 debug と setBreakpoint．現在の関数は既存パッケージに対するソースの幾つかを修正するケースを直接に目指しているが，同様にソースにデバッグコードを挿入することにも使える (もし含まれるデバッグが自明ではない時より有用である)．詳細節で述べられたように，ソースファイルはオリジナルのパッケージソース中のものと同じである必要はない．

Examples

```
## Not run:
## パッケージ P0 がソースファイル "all.R" を持つと仮定する
## 先ずソースを評価し，それから summary() 用のメソッドの
## 改訂バージョンを挿入する
  env <- insertSource("./P0/R/all.R", package = "P0",
     methods = "summary")
## そしてソースからのバージョンをトレース
## してメソッドの一つをテストする
  trace("summary", signature = "myMat", browser, edit = env)
## テスト後に browser() 呼び出しを取り去るがソースは残す
  trace("summary", signature = "myMat", edit = env)
## そしてソースファイルを再評価せずに
## 全ての(他の)改訂した関数とメソッドを挿入する．
```

```
## パッケージ名はオブジェクトの env に含まれる.
  insertSource(env)

## End(Not run)
```

---

findClass　　クラスを用いた計算

Description

クラス定義を見つけ操作する関数.

Usage

```
isClass(Class, formal=TRUE, where)

getClasses(where, inherits = missing(where))

findClass(Class, where, unique = "")

removeClass(Class, where,
            resolve.msg = getOption("removeClass.msg", default=TRUE))

resetClass(Class, classDef, where)

sealClass(Class, where)
```

Arguments

Class	クラス名の文字列. 関数は普通文字列の代わりにクラス定義を取る. 特定のパッケージ中で定義されたクラスに制限するには文字列の packageSlot を設定する.
where	その中で定義を修正したり削除する environment. 既定では呼び出し関数のトプレベル環境 (通常の計算に対しては大局的環境であるが, パッケージに対してはソース中のパッケージの環境下名前空間.) クラス定義を探索する際, where はどこで探すかを定義し, 既定ではこの関数を呼び出したものの環境か名前空間から探す.
formal	形式的定義が必要かどうかをを指示する logical.
unique	もし findClass がクラスのユニークな位置を期待するならば, unique は探索の目的を説明する文字列 (そして警告やエラーメッセージに使われる). 既定では, 多重位置が可能で関数は常にリストを返す.
inherits	getClasses の呼び出し中で, 返り値は where の全ての親環境を含むか, それともその環境だけか? 既定ではもし where が省略されていれば TRUE で, さもなければ FALSE.
resolve.msg	もし Class が複数の名前空間に見つかり一つが選ばれるならば, R がその決定を message() するかどうかを指示する logical.
classDef	resetClass に対して, オプションのクラス定義 (しかし普通 Class はクラス定義で, classDef) を省略するほうが好ましい.

Details

これらは形式的なクラス定義をテストし操作する関数である．以下に簡単な解説が与えられる．紹介とより詳細は参考文献を見よ．

`removeClass`: このクラスの定義を，もしこの引数が与えられていれば環境 `where` から取り除く；さもなければ `removeClass` は `removeClass` の呼び出しのトップレベル環境から初めて定義を探し，(最初に)見つかった定義を取り除く．

`isClass`: これは形式的に定義されたクラスの名前か? (引数 `formal` は互換性のためにあり無視される．)

`getClasses`: `where` について定義された全てのクラスの名前．もし引数無しで呼び出されると，呼び出し関数から可視の全てのクラス (もしトップレベルから呼び出されると，検索リスト上にある任意の環境中の凡てのクラス)． `inherits` 引数は特定の環境とその親を探索するのに使うことができるが，通常は既定の設定が希望するものである．

`findClass`: `Class` のクラス定義が見つかる環境または検索リスト上の位置のリスト．もし `where` が提供されると，これはそこで探索が行われる環境 (または名前空間)である；さもなければ呼び出しのトップレベルの環境が使われる．もし `unique` が文字列として提供されると， `findClass` は単一の環境下位置を返す．既定では，これは常にリストである．呼び出し関数は，例えば，最初の要素を `get` のような関数に対する位置や環境として選択すべきである．

もし `unique` が文字列として提供されると， `findClass` はもし一つより多くの定義が可視なら警告を出す(呼び出しの目的を特定する文字列を使い)，そしてもし定義が見つからなければエラーを生成する．

`resetClass`: クラスの内部定義をリセットする．クラスの完全な定義が `setClass` へのオリジナルな呼び出し中で指定された表現とスーパークラスから再計算される．

この関数はクラス定義のアスペクトが変更された時に呼び出される．このクラスが拡張するクラスの定義を変更する場合はそれを明示的に呼び出す必要がある． (しかしセッションの途中でそれを行うことは，既存のオブジェクトを不適切化するかもしれないので危険である).

`sealClass`: 指定クラスの現在の定義をそれ以上の変更を防ぐために封印する．呼び出し `setClass` 中のクラスを封印することは可能であるが，時として更なる変更がなされる必要ががある (例えば `setIs` の呼び出しを用いて)．もしそうなら，全ての関連する変更がなされた後に `sealClass` を呼び出す．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョン．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョン．)

See Also

`setClassUnion`, `Methods`, `makeClassRepresentation`

findMethods 総称的関数に対して定義されたメソッドの記述

Description

関数 findMethods は総称的関数に対するテーブル中で定義されているメソッド (メソッドの選択に対して使われるような)を吟味や表示のためにリストに変換する．リストは実際はクラス listOfMethods に由来する (下のクラスを説明する節を見よ).

リストはもし引数が提供されていれば環境 where 中で定義されたメソッドに限られ，そしてもし引数が提供されていればメソッドシグネチャ中に一つまたはそれ以上の classes で指定されたで定義されたものを含むもメソッドに限られる．

メソッド選択適用に対して使われる実際のテーブル (環境)を見るのは getMethodsForDispatch を呼び出す． findMethods が返すリストの名前はテーブル中のオブジェクトの名前である．

関数 findMethodSignatures はその行が対応するメソッドのシグネチャからのクラス名がである文字列行列である；これは findMethods が返すリストからか， findMethods と同じ引数を与えてそうしたリスト自体を計算して操作する．

関数は hasMethods TRUE か FALSE を，関数 f に対する空でないメソッドのテーブルが環境中にあるかまたは検索位置 where (またはもし where が欠損していれば一般的に総称的関数に対して) にあるかに応じて返す．

既定関数 getMethods は findMethods に対するより古い代替物で，以前にメソッド選択適用に対して使われたクラス MethodsList のオブジェクトの形式の情報を返す．このオブジェクトのクラスは一般に廃止予定で将来の R のバージョンでは無くなるため，これは推奨できない．

Usage

```
findMethods(f, where, classes = character(), inherited = FALSE,
      package = "")

findMethodSignatures(..., target = TRUE, methods = )

hasMethods(f, where, package)

## \code{table = FALSE} に対しては 2010 に廃止予定になり 2015 に廃止された
getMethods(f, where, table = FALSE)
```

Arguments

f	総称的関数またはそれの名前の文字列.
where	オプションで，メソッドのメタデータを探索する環境または検索リスト上の位置. もし where が欠損していると，findMethods は総称的関数自体中のメソッドの現在のテーブルを使い，そして hasMethods は検索リスト中のどこかのメタデータを探索する.
table	getMethods の呼び出し中でもし TRUE ならば返り値は選択適用に対して使われるテーブルであり，現在までに発見された継承メソッドを含む.内部的に使われるが，既定の結果は今は使われない mlist オブジェクトであるため，既定はいつか変更される可能性が高い.

classes もし提供されると，シグネチャが提供されたクラスの少なくとも一つを含むメソッドだけが返される値に含まれる．

inherited 論理値フラグ；もし TRUE ならば，継承も直接定義されたものも全てのメソッドのテーブル使われる；さもなければ明示的に定義されたメソッドだけ．オプションの TRUE は where が欠損している時だけ意味がある．

... findMethodSignatures の呼び出し中で， findMethods に与えられるかもしれない任意の引数．

target findMethodSignatures へのオプションのフラグ；もし TRUE ならば，使われるシグネチャは目的のシグネチャである (メソッドがそれに対して選択されるクラス)；もし FALSE ならば，それらは定義されているシグネチャになる．違いは inherited が TRUE の時だけ意味を持つ．

methods findMethodSignatures の呼び出し中では，オプションのメソッドリストで，おそらく先の findMethods の呼び出しにより返されたもの．もし欠損していれば，この関数は ... 引数で呼び出される．

package hasMethods の呼び出し中では，総称的関数に対するパッケージ名 (例えばプリミティブ関数に対しては "base")．もし欠損していればこれは，もしあれば，関数名の "package" 属性か，または総称的関数のパッケージスロットからか推測される． 'Details' を見よ．

Details

定義されたメソッドのテーブルを総称的関数か where で指定された環境中に保管されたメタデータオブジェクトから取得する．getMethods の呼び出し中では，テーブル中の情報は table 引数がある場合を除き，返り値を生成するために上で説明されたように変換される．

他の関数と違い，hasMethods はこの名前の総称的関数が現在見つからなくても使うことが出来る．この場合 package は引数として提供されるか，またはパッケージ名はメソッドテーブルの特定化の一部であるので f の属性として含まれる．

メソッドのリストに対するクラス

クラス "listOfMethods" はメソッドをメソッド定義の名前付きリストとして返す (またはプリミティブ関数，下のスロットの解説を見よ)．名前はそこからメソッドの選択適用が計算された環境中の対応するオブジェクトの保管に使われた文字列である．現在の実装はメソッドシグネチャ中の対応するクラスの名前を使い，もしシグネチャ中に複数の引数が含まれていれば "#" で分離される．

スロット

.Data: クラス "list" のオブジェクト．メソッド定義．
総称的関数はプリミティブ関数に対応するときは，これらはプリミティブ関数自体を既定メソッドとして含むかもしれないことを注意する． (基本的にプリミティブ関数は現在メソッド定義として拡張できない普通ではない R オブジェクトであるため．) 他の情報を導くために返されたリストを使う計算はこの可能性を考慮する必要がある．例えば findMethodSignatures の実装を見よ．

arguments: クラス "character" のオブジェクト．総称的関数のシグネチャ中の形式的引数の名前．

signatures: クラス "list" のオブジェクト．個別のメソッドのシグネチャのリスト．これは現在 names を分離記号 "#" で分割した結果である．

もしオブジェクトが findMethods により返された時のようにテーブルから構成されていれば，シグネチャは全て同じ長さを持つ．しかしながら，一般性のために文字列行列よりもリストが使われる．下の例のような findMethodSignatures の呼び出しは常に行列形式に変換する．

generic: クラス "genericFunction" のオブジェクト．これらのメソッドに対応する総称的関数．このスロットを関数参照を許すように一般化する計画がある．

names: クラス "character" のオブジェクト．注意された名前は "#" で分離されたクラス名である．

拡張

クラス "namedList"，直接に．

クラス "list"，クラス "namedList" を使い，距離 2.

クラス "vector"，クラス "namedList" を使い，距離 3.

See Also

showMethods, selectMethod, Methods

Examples

```
mm <-  findMethods("Ops")
findMethodSignatures(methods = mm)
```

---

fixPre1.8 バージョン *1.8* 以前の *R* でセーブされたオブジェクトをフィックスする

Description

バージョン 1.8.0 の Rからオブジェクトのクラスはクラスが定義されているパッケージの識別子を含んでいる．関数 fixPre1.8 はこの情報を欠いているオブジェクトをフィックスし再付値する (典型的にそれらは以前のバージョンの R を用いてセーブされたファイルからロードされているため).

Usage

```
fixPre1.8(names, where)
```

Arguments

names	フィックスまたは再付値される全てのオブジェクトの名前の文字列ベクトル.
where	オブジェクトとクラス定義をそこから探索する環境．既定では fixPre1.8 呼び出しのトップ環境で，関数が対話的に使われていれば大局的環境.

Details

名前のオブジェクトがそれが見つかった場所に保存される．そのクラス属性は R 1.8 が要求する完全な形式に変更される；さもなければオブジェクトの内容は変更されるべきではない．

オブジェクトは以下の条件が成り立つ時だけフィックスされ再付値される：

1. 名前のオブジェクトが存在する．
2. それは定義されたクラスからのものである (実際のクラス属性を持たない基本データタイプではない)．
3. オブジェクトは以前のバージョンの R からのものに見える．
4. クラスが現在定義されている．
5. オブジェクトは現在のクラス定義と一貫性がある．

二番目を除くどれかの条件が成り立たなければ警告が出る．

`fixPre1.8` は現在クラス属性中の変更だけをフィックスすることを注意する．特に，それは以前のバージョンの R でインストールされたバイナリーバージョンのパッケージを，もしそれらが相容れない特性をもっていてもフィックスしない．そうしたパッケージはソースから再インストールされなければならず，これは R のバージョンの主要な変更がされた場合は常に賢明なアプローチである．

Value

実際は再付値された全てのオブジェクトの名前．

---

genericFunction-class　総称的関数オブジェクト

Description

総称的関数(`genericFunction` を拡張するオブジェクト)は拡張されたオブジェクトで，この関数に対してメソッドを作り選択適用するのに使われる情報を含む．それらはまた関数やメソッドに関連するパッケージを特定する．

クラスからのオブジェクト

総称的関数は `setGeneric` や `setGroupGeneric` により，して間接的に `setMethod` により作られ付値される．

期待されるであろうように `setGeneric` と `setGroupGeneric` はそれぞれクラス `"genericFunction"` と `"groupGenericFunction"` のオブジェクトを作る．

スロット

`.Data`: クラス `"function"` のオブジェクト．総称的関数の関数定義で，普通 `standardGeneric` の呼び出しとして自動的に作られる．

`generic`: クラス `"character"` のオブジェクトで，総称的関数の名前．

`package`: クラス `"character"` のオブジェクトで，関数定義が属するパッケージ名 (そして必ずしも総称的関数が保存されている場所では無い)．もしパッケージが `setGeneric` の呼び出しで明示的に指定されないと，それは普通対応する非総称的な関数が存在するパッケージである．

group: クラス "list" のオブジェクトで，この総称的関数が属するグループかグループ群．既定では空．

valueClass: クラス "character" のオブジェクト；もし空の文字列でなければ，一つまたはそれ以上のクラスを特定する．この関数に対する全てのメソッドがこれらのクラス(またはそれらを拡張するクラス)からのオブジェクトを返すことが保証する．

signature: クラス "character" のオブジェクトで，この総称的関数に対するメソッドのシグネチャ中に登場できる形式的引数名のベクトル．既定ではこれは ... に対するもの除く全ての形式的引数である．効率性のためには順序が肝要である：メソッドを指定する際に最も普通に使われる引数が最初に来るべきである．

default: クラス"optionalMethod" のオブジェクト (クラス "function" と "NULL" の合併)で，もしあればこの関数に対する既定メソッドを含む．メソッドの選択適用の初期化するために自動的に生成され使われる．

skeleton: クラス"call" のオブジェクトで，メソッドの選択適用で内部的に使われるスロット．それを直接使うことを期待しない．

拡張

クラス "function"，データ部分から．
クラス "OptionalMethods"，クラス "function" により．
クラス "PossibleMethod"，クラス "function" により．

メソッド

総称的関数オブジェクトは形式的メソッドを作り選択適用する際に使われる；オブジェクトからの情報はメソッドリストオブジェクトを作り，この総称的関数に対する既存のメソッドを併合し更新するのに使われる．

GenericFunctions 総称的関数の管理のためのツール

Description

ここでドキュメント化される関数は総称的関数に関連するメソッドの集まりを管理し，同様に総称的関数自体に関する情報を提供する．

Usage

```
isGeneric(f, where, fdef, getName = FALSE)
isGroup(f, where, fdef)
removeGeneric(f, where)

dumpMethod(f, signature, file, where, def)
findFunction(f, generic = TRUE, where = topenv(parent.frame()))
dumpMethods(f, file, signature, methods, where)
signature(...)

removeMethods(f, where = topenv(parent.frame()), all = missing(where))

setReplaceMethod(f, ..., where = topenv(parent.frame()))

getGenerics(where, searchForm = FALSE)
```

Arguments

f	関数の名前である文字列.
where	オブジェクトを探す環境，名前空間，または検索リスト位置．既定では呼び出し関数のトップレベル環境から始まり，典型的には大局的環境 (つまり検索リストを使う)，または呼び出しが由来するパッケージの名前空間．これらの関数のどれかを間接的に呼び出す際はこの引数を提供することが重要である．パッケージの名前空間と共には既定はそうした呼び出し中では誤る可能性が高い.
signature	関連するメソッドのクラスシグネチャ．シグネチャは名前付きか名前無しの文字列のベクトルである．もし名前付きなら，名前は総称的関数に対する形式的引数名でなければならない．シグネチャは総称的関数のシグネチャスロット中で指定された引数にマッチされる (setMethod のドキュメントの詳細節を見よ). dumpMethods への signature 引数は無視される (これは過去の実装中で内部的に使われていた).
file	そこにメソッド定義をダンプするファイルかコネクション.
def	メソッドを定義する関数オブジェクト；もし省略されるとシグネチャに対応する現在のメソッド定義.
...	シグネチャを形作る名前付きか名前無しの引数.
generic	関数をテストしたり見つける際，総称的関数も含めるべきか．非総称的関数だけを得るには FALSE と置く.
fdef	オプション，総称的関数定義. isGeneric への呼び出しでは普通省略される.
getName	もし TRUE なら isGeneric は総称的関数の名前を返す．既定では TRUE を返す.
methods	ダンプされるメソッドを含むメソッドオブジェクト．既定ではこの総称的関数に対して定義されるメソッド (オプションで指定された where 位置で).
all	removeMethods 中で，全て (既定)か最初に見つかったメソッドが消去されるかを指示する論理値.
searchForm	getGenerics 中で，もし TRUE ならば返された結果の package スロットは search() で使われる形式で，さもなければ単純なパッケージ名(例えば "package:base" 対 "base").

関数の要約

isGeneric: f という名前の関数があるか，もしあれば総称的か?

getName 引数は関数が名前を関数定義から見つけることを可能にする．もしこれが TRUE なら総称的関数の名前が返される．またはもしこれが総称的関数定義でなければ FALSE を返す.

プリミティブ関数に対する isGeneric と getGeneric の挙動は少々異なる．これらの関数はそれらに対してメソッドが定義されているかどうかにかかわらず，形式的関数オブジェクトとしては存在しない (効率性と歴史的理由から) isGeneric の呼び出しはこのプリミティブ関数に対してメソッドが現在の検索リストや指定された位置 where のどこかに定義されているかどうかを知らせる．対照的に getGeneric の呼び出しは，それに対して如何なるメソッドも定義されていなくても，その関数に対する総称性を返す.

removeGeneric, removeMethods: この名前の総称的関数に対する全てのメソッドを取り除く．加えて removeGeneric は関数自体を取り除く； removeMethods は既定メソッドであった非総称的関数を回復する．もし既定メソッドが無ければ removeMethods はメソッドが無い総称的関数のままにする．

standardGeneric: 総称的関数 f に対する現在の関数呼び出しからのメソッドを選択適用する． standardGeneric を対応する総称的関数の本体内以外から呼び出すことはエラーである．

standardGeneric は効率性のため **base** パッケージ中のプリミティブ関数であるが，それが本来属するここでドキュメント化されていることを注意する．

dumpMethod: この総称的関数とシグネチャに対するメソッドをダンプする．

findFunction: name に対する関数オブジェクトが存在する検索リスト上の位置か現在のトップレベルの環境のリストを返す．返り値は 常にリストであり，最初の要素を関数の最初の可視のバージョンにアクセスするために使う．例を見よ．

注意： find を mode="function" で使うよりはこれを使う，前者は内容がそれほど無く，その正規表現の使用法から幾つかの微妙なバグを持つ．また library の呼び出しを使ってパッケージを付加した際にパッケージに対するコード中でも正しく動作する．

dumpMethods: この総称的関数に対する全てのメソッドをダンプする．

signature: 総称的関数の引数にマッチされたクラスの名前付きリストを返す．

getGenerics: where にメソッドが定義されている総称的関数の名前を返す；この引数は環境でも検索パスへの添字でも良い．既定では全ての検索パスが探される．

メソッドの定義はパッケージの修飾子と共に保管されている；例えば関数 "initialize" に対するメソッドは異なったパッケージ上のその名前の二つの異なった関数を参照するかもしれない．メソッドリストオブジェクトに対応するパッケージ名は返されるオブジェクトのスロット package 中に含まれている．返される名前の書式は searchForm の値に応じて平易(例えば "base") でも検索リスト("package:base")中で使われる形式でも良い．

詳細

setGeneric: もしすでにこの名前の非総称的関数が存在すれば， def が提供されない限りそれは総称的関数の定義に使われる．そして現在の関数は総称的関数に対する既定メソッドになる．

もし def が提供されると，これは総称的関数を定義しそして既定メソッドはない(関数が全てのオブジェクトの意味のある部分集合に対してだけ利用可能であるべきなら，しばしば良い特徴である)．

引数 group と valueClass は S-Plus との互換性のために保持されているが現在使われない．

isGeneric: もし fdef 引数が提供されると，これが総称的関数の定義とされ，それが本当に総称的関数かどうかをテストする． f は総称的関数の名前とされる． (この引数は S-Plus では利用できない．)

removeGeneric: もし where が提供されると，検索リストのこの要素のバージョンを単に取り除く；さもなければ最初にであったバージョンが取り除かれる．

standardGeneric: 総称的関数は普通それらの本体全体として standardGeneric への呼び出しを持つべきである．しかしながら，任意の他の計算も又行える．

通常の setGeneric (直接または setMethod の呼び出し経由で) は standardGeneric への呼び出しを持つ関数を作る．

dumpMethod: 結果のソースファイルはメソッドを再生性する．

`findFunction`: もし `generic` が `FALSE` ならば，総称的関数を無視する．

`dumpMethods`: もし `signature` が提供されるとこの初期シグネチャにマッチするメソッドだけがダンプされる． (この特徴は S-Plus には無い：互換性のためには使わないこと．)

`signature`: `signature` を使う利点はどの引数を考えているかについてのチェックと並んで，メソッド指定に於いてより明快なドキュメント化を提供することである．加えて `signature` は要素の各々が単一の文字列であることをチェックする．

`removeMethods`: もし `f` が総称的関数であれば `TRUE` を返す． さもなければ `FALSE` を(黙って)返す．

もし既定メソッドがあれば，関数はこの定義を持つ単純な関数として再付値される． さもなければ，総称的関数にとどまるが既定メソッドは無い (従ってそれへの任意の呼び出しはエラーを発生する)． どちらの場合も引き続く `setMethod` への呼び出しは以前のような同じ総称的関数を一貫性を持って再確立する．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version.)

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

`getMethod` (`selectMethod` に対しても)， `setGeneric`, `setClass`, `showMethods`

Examples

```
require(stats) # lm に対して

## 関数 "myFun" を得る -- もし 0 かまたは > 1 バージョンが可視なら：
findFuncStrict <- function(fName) {
  allF <- findFunction(fName)
  if(length(allF) == 0)
    stop("No versions of ",fName," visible")
  else if(length(allF) > 1)
    stop(fName," is ambiguous: ", length(allF), " versions")
  else
    get(fName, allF[[1]])
}

try(findFuncStrict("myFun"))# エラー：どのバージョンも無い
lm <- function(x) x+1
try(findFuncStrict("lm"))#    エラー：二つのバージョン
findFuncStrict("findFuncStrict")# 丁度一つのバージョン
rm(lm)



## メソッドのダンプ ----------------------------------

setClass("A", representation(a="numeric"))
setMethod("plot", "A", function(x,y,...){ cat("A meth\n") })
dumpMethod("plot","A", file="")
## Not run:
setMethod("plot", "A",
```

```
function (x, y, ...)
{
    cat("AAAAA\n")
}
)

## End(Not run)
tmp <- tempfile()
dumpMethod("plot","A", file=tmp)
## ここで取り除き，ダンプを構文解析できるかどうかを見る
stopifnot(removeMethod("plot", "A"))
source(tmp)
stopifnot(is(getMethod("plot", "A"), "MethodDefinition"))

## dumpMethods() と同じ：
setClass("B", contains="A")
setMethod("plot", "B", function(x,y,...){ cat("B ...\n") })
dumpMethods("plot", file=tmp)
stopifnot(removeMethod("plot", "A"),
          removeMethod("plot", "B"))
source(tmp)
stopifnot(is(getMethod("plot", "A"), "MethodDefinition"),
          is(getMethod("plot", "B"), "MethodDefinition"))
```

---

getClass　　クラス定義を得る

Description

クラスの定義を得る.

Usage

```
getClass   (Class, .Force = FALSE, where,
            resolve.msg = getOption("getClass.msg", default=TRUE))
getClassDef(Class, where, package, inherits = TRUE,
            resolve.msg = getOption("getClass.msg", default=TRUE))
```

Arguments

Class	クラス名の文字列で，しばしば下の "package" 属性を持つ.
.Force	もし TRUE ならばクラス名が未定義なら NULL を返す；さもなければ未定義のクラスはエラーになる.
where	定義の探索をそこから始める環境；既定ではトップレベルの環境 (大局的環境)から始まり検索パスに沿って進行する.
package	定義を含むと主張されるパッケージの名前. もしこれが空でない文字列ならばクラスを探す最初の場所として where の代わりに使われる. パッケージはロードされていなければならないが付加されている必要はないことを注意する. 既定では Class 引数のパッケージ属性がもしあれば使われる. もし Class があるオブジェクトに対する class(x) に由来すれば普通パッケージ属性がある.

inherits 論理値；クラス定義は任意の囲み環境と同様にキャッシュから検索されるべきか? もし FALSE ならば環境 where 中の定義だけが返される．

resolve.msg Class が複数の名前．中で見つかり一つが選択された時 R がその決定を message() すべきかどうかを指示する logical．

Details

クラス定義はパッケージの名前空間かそれらが定義された他の環境中のメタデータオブジェクト中に保管される．パッケージがロードされた時，パッケージ中のクラス定義は内部テーブルにキャッシュされる．従って，inherits が FALSE でない限り getClassDef の殆どの呼び出しはキャッシュ中でクラスを見つけるかそれを全く見つけることが出来ない． inherits が FALSE の場合は package か where で定義された環境だけが探索される．

クラスのキャッシュは同じクラス名の別個の環境への多重定義を許すが，勿論パッケージ属性かパッケージ名が呼び出し中で与えられなければならないという制限付きである．

Value

クラスを定義するオブジェクト．もしクラス定義が見つからなければ getClassDef は NULL を返すが，getClassDef を呼び出す getClass はエラーを生成するか，もし .Force が TRUE ならばクラスに対する単純な定義を返す．後者の場合は内部的に使われるが，ユーザのコード中では典型的に無意味である．

null でない値はクラス classRepresentation のオブジェクトである．全ての合理的な目的に対して，これを修正のためではなく情報の取り出しにだけ使うこと：クラス定義の作成や修正には関数 setClass と setIs を使う．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルな S4 バージョンに対して．)

See Also

Classes, setClass, isClass.

Examples

```
getClass("numeric") ## 組み込みクラス

cld <- getClass("thisIsAnUndefinedClass", .Force = TRUE)
cld ## NULL プロトタイプ
## もし本当に関心があれば：
utils::str(cld)
## これらはエラーを生成するものの：
try(getClass("thisIsAnUndefinedClass"))
try(getClassDef("thisIsAnUndefinedClass"))
```

getMethod メソッド定義を得る，テストする

Description

与えられた総称的関数とシグネチャに対応するメソッドを探す関数．関数 getMethod と selectMethod はメソッドを返す；関数 existsMethod と hasMethod はその存在をテストする．どちらのケースも最初の関数は直接の定義だけを受け入れ，そして二番目は継承を用いる．全てのケースで，探索は総称的関数自体中か引数 where で指定されるパッケージ/環境中で行われる．

関数 findMethod はこの関数とシグネチャに対するメソッドを含む検索パス (もしくは引数 where で指定されるパッケージ) 中のパッケージを返す．

Usage

```
getMethod(f, signature=character(), where, optional = FALSE,
          mlist, fdef)

existsMethod(f, signature = character(), where)

findMethod(f, signature, where)

selectMethod(f, signature, optional = FALSE, useInherited =,
             mlist = , fdef = , verbose = , doCache = )

hasMethod(f, signature=character(), where)
```

Arguments

f	総称的関数またはその文字列名．
signature	引数 f にマッチするクラスのシグネチャ．下の詳細を見よ．
where	メソッドを探す位置または環境：既定では総称的関数自体中に定義されたメソッドのテーブルが使われる．
optional	もし selectMethod 中の選択が適正なメソッドを見つけられなければこの引数が TRUE で無い限りエラーが生じる．その場合，如何なるメソッドもマッチしなければ NULL が返される．
mlist, fdef, useInherited, verbose, doCache	内部的に使われる getMethod と selectMethod へのオプションの引数．これらは避けたほうが良い：あるものは期待通りに動作するがしないものもあり，どれもが関数の普通の使用では必要とされない．

Details

引数 signature は総称的関数の形式的引数に対応するクラスを指定する；正確に言えば，総称的関数の signature スロットに対応するもの．引数はクラスを特定する文字列ベクトルであってよく，名前付きでも無しでも良い．もし提供されれば名前は総称的関数のシグネチャ中に含まれる形式的引数の名前にマッチする．このシグネチャは普通 ... を除く全ての引数である．しかしながら，総称的関数は許される引数のサブセットだけ，または異なった順序で取った引数を用いて指定することが出来る．

もしそのシグネチャに対して何か特別なことを行う総称的関数を扱っているのならば，混乱を避けるためにシグネチャ中の引数に名前を付けることは良いアイデアである．どんな場合も，シグネチャ中の要素は関数呼び出し中の引数マッチング中で使われるのと同じ規則で形式的シグネチャにマッチされる (match.call を見よ).

シグネチャ中の文字列はクラス名， "missing" または "ANY" であって良い．メソッド選択におけるこれらの意味については Methods を見よ．シグネチャ中で提供されない引数は暗黙のうちにクラス "ANY" に対応する；特に，空のシグネチャを与えることは既定メソッドを探すことを意味する．

getMethod の呼び出しは特定の関数とシグネチャに対するメソッドを返す．他の get 関数と同様に，引数 where は関数がどこを探すか (既定では検索リストのあらゆる所)を制御し，引数 optional はもしメソッドが見つからない時 NULL を返すかエラーを発生するかを制御する．メソッドの探索は継承を利用しない．

関数 selectMethod は又関数とシグネチャを与えてメソッドを探すが，メソッド選択適用機構の完全な利用を行う，つまり対応するシグネチャに対する選択適用に於けるように継承メソッドとグループ総称的関数を考慮するが，条件付き継承は使われない．getMethod と同様に，selectMethod はもしメソッドが見つからなければ，引数 optional に依存して， NULL を返すかエラーを生成する．

関数 existsMethod と hasMethod はメソッドが見つかるか否かで TRUE か FALSE を返し，最初は getMethod に対応し(継承無し)二番目は selectMethod に対応する．

Value

selectMethod または getMethod の呼び出しは，もし見つかれば選択されたメソッドを返す． (このクラスは function を拡張するため，結果が希望するものであれば関数を直接に使うことが出来る．) さもなければ，もし optional が FALSE ならばエラーが発生し，そしてもし optional が TRUE ならば NULL が返される．

返されるメソッドオブジェクトはプリミティブ関数に対する既定メソッドがそれ自体プリミティブであることが必要で無い限り MethodDefinition オブジェクトである．従って検索の失敗に対する唯一の信頼できるテストは is.null() である．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

メソッドの選択適用の詳細は Methods；メソッドと総称的関数オブジェクトを操作する他の関数については GenericFunctions；メソッド定義を表現するクラスについては MethodDefinition.

Examples

```
setGeneric("testFun", function(x)standardGeneric("testFun"))
setMethod("testFun", "numeric", function(x)x+1)
hasMethod("testFun", "numeric")
## Not run: [1] TRUE
hasMethod("testFun", "integer") #継承
## Not run: [1] TRUE
existsMethod("testFun", "integer")
```

```
## Not run: [1] FALSE
hasMethod("testFun") # 既定メソッド
## Not run: [1] FALSE
hasMethod("testFun", "ANY")
## Not run: [1] FALSE
```

---

getPackageName 与えられたパッケージに関連する名前

Description

下の関数は特定の環境か検索リスト上の位置に関連するパッケージか，または特定の関数を含むパッケージのパッケージを作る．これらは主として複数パッケージ上のオブジェクトを区別するために必要な計算をサポートするために使われる．

Usage

```
getPackageName(where, create = TRUE)
setPackageName(pkg, env)

packageSlot(object)
packageSlot(object) <- value
```

Arguments

where	希望のパッケージに関連する環境または検索リスト上の位置.
object	文字列名を与えるオブジェクト，プラスオブジェクトが存在するパッケージ.
value	パッケージ名.
create	フラグ，もし推測できなければパッケージ名を作るか? もし TRUE で空でないパッケージ名が見つからなければ，現在の日付と時間がパッケージ名に使われ，そして警告が出る．もし環境がロックされていなければ作られた名前は環境中に保管される.
pkg, env	pkg 中の文字列を環境 env 中のクラスとメソッド定義を設定する全ての計算に対する内部パッケージ名にする.

Details

パッケージ名は普通パッケージのロード中に INSTALL スクリプトか library 関数によりインストールされる． (現在，名前は .packageName オブジェクトとして保存されるが将来の保証はない．)

Value

getPackageName はパッケージの文字列名を返す (検索リスト中で見つかった余分の "package:" 無しで).

packageSlot はパッケージの名前スロットを返す，または設定する (現在属性で形式的なスロットではないがいつか変更されるかもしれない).

setPackageName はさもなければパッケージ名を持たない環境中にパッケージ名を確立するのに使うことが出来る．これは任意の環境中にクラス/メソッドを作るのを許すが，しかし普通は標準的な R のプログラミングツール (package.skeleton 等)でパッケージを作るのが好ましい．

See Also

search, packageName

Examples

```
## 次の全ては普通 "base" を返す
getPackageName(length(search()))
getPackageName(baseenv())
getPackageName(asNamespace("base"))
getPackageName("package:base")
```

---

hasArg 呼び出し中の引数を探す

Description

もし name が呼び出し中の引数，関数への形式的引数かまたは ... の成分，に対応すれば TRUE を，そしてさもなければ FALSE を返す．

Usage

```
hasArg(name)
```

Arguments

name	引用化されていない名前か文字列としての可能な引数の名前．

Details

例えば表現式 hasArg(x) は二つの例外を次の除いて !missing(x) に似ている．先ず hasArg は x が呼び出し関数の形式的引数ではないが ... のそれであれば名前が x の引数を探す．次に hasArg はもし名前を引数として与えられれば決してエラーを発生しないが missing(x) は x が形式的引数でなければエラーを発生する．

Value

上で説明されたように常に TRUE か FALSE．

See Also

missing

Examples

```
ftest <- function(x1, ...) c(hasArg(x1), hasArg("y2"))

ftest(1) ## c(TRUE, FALSE)
ftest(1, 2)  ## c(TRUE, FALSE)
ftest(y2 = 2)   ## c(FALSE, TRUE)
ftest(y = 2)    ## c(FALSE, FALSE) (部分マッチなし)
ftest(y2 = 2, x = 1)  ## c(TRUE, TRUE) 部分的に x1 にマッチ
```

---

implicitGeneric 総称的関数の暗黙のバージョンを管理する

Description

現在総称的関数でない関数を一貫性のある総称的バージョンに強制するのに使われる暗黙の総称的関数を作るまたはアクセスする．関数 implicitGeneric() は暗黙の総称的バージョンを返し， setGenericImplicit() は総称的関数を暗黙化し， prohibitGeneric() は関数を総称的にするのを防止し，そして registerImplicitGenerics() は現在のセッションのキャッシュテーブルに暗黙の総称的定義のセットを保存する．

Usage

```
implicitGeneric(name, where, generic)
setGenericImplicit(name, where, restore = TRUE)
prohibitGeneric(name, where)
registerImplicitGenerics(what, where)
```

Arguments

name	関数の文字列名.
where	暗黙の総称的関数を登録するパッケージか環境．この関数を自分自身のトップレベルから使う時はこの引数は普通省略可能である(そしてそうあるべきである).
generic	オプションのキャッシュされる総称的関数定義であるが，普通省略可能であるされる．詳細節を見よ.
restore	現在の後に非総称的な関数のバージョンを回復すべきか.
what	registerImplicitGenerics() に対して．登録される暗黙の総称的関数のオプションのテーブルであるが，ほとんど常に無視される．下の詳細節を見よ.

Details

一つのパッケージに保管された関数のバージョンを使い，複数のパッケージが同じ関数のメソッドを定義することがある．これら全ての関数はユーザが関数を呼び出す時統括され統一的に選択適用されるべきである．一貫性のために関数の総称的バージョンはユニークな定義を持つべきである (メソッドのシグネチャ中に同じ引数，値クラス等のオプションのスロットに対する同じ値，そして関数自体の同じ標準的か非標準的な定義).

オリジナルの関数がすでに S4 総称的であれば何の問題もない．暗黙の総称的関数機構は関数を所有するパッケージ中のバージョンが総称的で無い時は一貫性を強要する．もし setGeneric() の呼び出しが別のパッケージ中の関数を総称的にするときは，機構は提案

される新しい総称的関数をその関数の暗黙の総称的バージョンと比較する．もし二つが一致すれば全てがうまく行く．もし一致せず，そして関数が別のパッケージに属していれば，新しい総称的関数はそのパッケージに関連付けられない．代わりに警告が出され，そして別個の総称的関数が作られ，そのパッケージスロットは関数の非総称的バージョンを所有するパッケージではなく現在のパッケージに設定される．その効果は新しいパッケージは依然としてこの関数に対するメソッドを定義するが，それは総称的関数の異なった定義を強制するため，他のパッケージ中のメソッドを共有しない．

ほとんどすべてのケースで取るべき正しい方法は関数の名前だけを与えて `setGeneric("foo")` を呼び出すことである；これは自動的に暗黙の総称的バージョンを使う．もしこのバージョンが気に入らなければ，最良の解決法は他のパッケージの所有者を説得同意させ関数の非既定的性質を定義するコードを挿入してもらうことである (たとえ所有者が `foo()` を既定で総称的にしたくなくても).

任意の関数に対して，暗黙の総称的関数型は `...` を除く全ての形式的引数がメソッドのシグネチャ中に許される標準的な総称的関数である．もしこれが関数に対する適当な総称的関数ならば，如何なるアクションも不要である．さもなければ，最良の機構は関数を所有するパッケージのコード中に総称的関数を設定し，そしてそれから暗黙の総称的関数を記録するために `setGenericImplicit()` を呼び出し，それから非総称的なバージョンを回復することである．例を見よ．

パッケージは暗黙の総称的関数に対するメソッドも同様に定義できる；暗黙の総称的関数が真の総称的関数にされると，これらのメソッドも含まれる．

メソッドを予め定義する以外に，非既定の暗黙の総称的関数を持つ普通の理由は非既定のシグネチャを提供することで， それに対する普通の理由はある引数の遅延評価を許すことにある．例を見よ．総称的関数のシグネチャ中の全ての引数は関数がメソッドの選択が必要な時に評価されていなければならない．(しかしこれらの引数は既定表現が定義されていても無くても欠損することが出来る；シグネチャ中の引数に対してすらも常に `missing(x)` を吟味することが出来る．)

もし自分の関数を誰かが総称的にすることを完全に拒否したければ `prohibitGeneric()` を呼び出す．

Value

関数 `implicitGeneric()` は暗黙の総称的関数定義を返す (そしてそれを構築しなければならないならその定義を最初にキャッシュする).

他の関数はそれらの副作用のために存在し有用なものは何も返さない．

See Also

`setGeneric`

Examples

```
### 関数 \link{with}() をどのように総称的関数にしたか:

## シグネチャ中に "data" だけがあるように
## with() を使いたい二つ目の引数，'expr' が文字通りに使われる．

## 内部的な 'methods' コードは次と同値なことを行うように
## with() を拡張していることを注意する
## Not run:
setGeneric("with", signature = "data")
## もし希望すればここで
```

```
## "with" に対するメソッドを予め定義することも出来た.

## 準備ができれば，総称的関数を暗黙バージョンとして保管し，
## オリジナルの setGenericImplicit("with") を回復する

## (この例は関数 with() が存在する時だけ動作するが，
## それは基本パッケージ中にある. )
## End(Not run)

implicitGeneric("with")
```

---

inheritedSlotNames　　スーパークラスから継承されたスロットの名前

Description

クラス(またはクラス定義，getClass を見よ)とクラス定義 classRepresentation)，“上”から継承されている名前を与える，つまりスーパークラスで，このクラス定義自体ではない.

Usage

```
inheritedSlotNames(Class, where = topenv(parent.frame()))
```

Arguments

Class	文字列または classRepresentation，つまり getClass に由来.
where	環境で，更に isClass と getClass に渡される.

Value

スロット名の文字列ベクトル，または NULL.

See Also

slotNames, slot, setClass 等.

Examples

```
.srch <- search()
library(stats4)
inheritedSlotNames("mle")

if(require("Matrix")) {
  print( inheritedSlotNames("Matrix") ) # NULL
  ## 他方で
  print( inheritedSlotNames("sparseMatrix") ) # --> Dim & Dimnames
  ##  つまり "Matrix" クラスを継承

  print( cl <- getClass("dgCMatrix") ) # 6つのスロット，等

  print( inheritedSlotNames(cl) ) # 6つ*全ての*スロットは継承されている
}
```

```
## Not run:


## 上で付加されたパッケージを切り離す：
for(n in rev(which(is.na(match(search(), .srch)))))
   try( detach(pos = n) )

## End(Not run)
```

---

initialize-methods　　あるクラスからの新しいオブジェクトを初期化するメソッド

Description

特定のクラスからのオブジェクトを作るための関数 new への引数はそのクラスへの関数 initialize に対するメソッドの定義によりそのクラスに対して特別に解釈される．このドキュメントは幾つかの既存のメソッドを解説し，そして新しい物をどのように書くかを概説する．

メソッド

signature(.Object = "ANY") initialize に対する既定メソッドは名前付きまたは名前無しの引数を取る．引数名はこのクラス定義中のスロット名でなければならず，そして対応する引数はスロットに対する適正なオブジェクトでなければならない (つまり，スロットに対して指定されたのとおなじクラスか，またはそのクラスのスーパークラス)．もしオブジェクトがスーパークラス由来ならば，それは厳格には強制変換されず，従って普通それはその現在のクラスを保つ (特に as(object, Class, strict = FALSE))．
名前無しの引数はこのクラスのオブジェクト，そのスーパークラスのオブジェクト，またはそのサブクラスの一つでなければならない (クラス，このクラスが拡張しているクラス，またはこのクラスを拡張しているクラスから)．もしオブジェクトがスーパークラス由来ならば，これは普通オブジェクト中のスロットのどれかを定義する．もしオブジェクトがサブクラスからならば，現在のクラスに強制変換される．
名前無し引数はそれらが登場する順序で最初に処理される．それから名前付きの引数が処理される．従って，スロットに対する明示的な値は常にスーパークラスかサブクラスから推測される任意の値を上書きする．

signature(.Object = "traceable") traceable を拡張するクラスのオブジェクトはデバッグのトレースを実装するために使われる (クラス traceable と trace を見よ)．
これらのクラスに対する initialize メソッドは def, tracer, exit, at, print という特殊な引数を取る．これらの最初はオリジナルな定義として使われるオブジェクトである (例えば関数)．他は trace への引数に対応する．

signature(.Object = "environment"),signature(.Object = ".environment") 環境に対する initialize メソッドは環境を初期化するために使われるオブジェクトの名前付きリストを取る． "environment" のサブクラスは ".environment" を通じて初期化メソッドを継承し，新しい環境を確保する追加の効果を持つ．もしそうしたサブクラスに対する自分自身のメソッドを定義するならば，既存のメソッドを callNextMethod を使って呼び出すか，または自分のメソッド中で環境を確保することを忘れない．環境は参照であり自動的には複製されないからである．

signature(.Object = "signature") これは内部的な使用専用のメソッドである．これはオプションの functionDef 引数で引数名を定義するsignature スロットを持つ総称的関数を提供する．詳細は Methods を見よ．

初期化メソッドを書く

初期化メソッドは他の言語中の生成関数に対応する一般的な機構を提供する．

initialize への引数は .Object と ... である．ほとんど常に initialize は new から呼び出され，直接にではない． .Object 引数はそうするとクラスからのプロトタイプオブジェクトである．

二つのテクニックがしばしば initialize メソッドに対して適当である：特殊な引数名と callNextMethod である．

(既定の)スロット名よりもユーザにより自然な引数名が欲しいかもしれない．これらは .Object (常に)と ...(オプション)に追加される自分のメソッド定義への形式的引数である．例えば，上で解説されたクラス "traceable" に対するメソッドは次のような形式の setMethod への呼び出しで作られる：

```
setMethod("initialize", "traceable",
  function(.Object, def, tracer, exit, at, print) ...
)
```

この例では，他の引数のどれもが意味を持たず，そして結果のメソッドは他の引数が与えられるとエラーを惹き起こす．

もし新しいクラスが別のクラスを拡張すると，このスーパークラス(特殊メソッドか既定)に対するメソッドを初期化したいかもしれない．例えば特殊引数 x を持つが，同時にユーザがスロットを独自に設定できるようにしたい自分のクラスに対するメソッドを定義したいと仮定する．もし x がスロット情報を上書きしたければ，メソッド定義の冒頭は次の様なものになるかもしれない：

```
function(.Object, x, ...) {
  Object <- callNextMethod(.Object, ...)
  if(!missing(x)) { # x に何かをする
```

また，最初に x を解釈しそれから次のメソッドを呼びだすことで，継承メソッドを上書きすることを選択できたであろう．

Introduction *Basic use of S4 Methods and Classes*

Description

The majority of applications using methods and classes will be in R packages implementing new computations for an application, using new *classes* of objects that represent the data and results. Computations will be implemented using *methods* that implement functional computations when one or more of the arguments is an object from these classes.

Calls to the functions setClass() define the new classes; calls to setMethod define the methods. These, along with ordinary R computations, are sufficient to get started for most applications.

Classes are defined in terms of the data in them and what other classes of data they inherit from. Section 'Defining Classes' outlines the basic design of new classes.

Methods are R functions, often implementing basic computations as they apply to the new classes of objects. Section 'Defining Methods' discusses basic requirements and special tools for defining methods.

The classes discussed here are the original functional classes. R also supports formal classes and methods similar to those in other languages such as Python, in which methods are part of class definitions and invoked on an object. These are more appropriate when computations expect references to objects that are persistent, making changes to the object over time. See ReferenceClasses and Chapter 9 of the reference for the choice between these and S4 classes.

Defining Classes

All objects in R belong to a class; ordinary vectors and other basic objects are built-in (builtin-class). A new class is defined in terms of the named *slots* that is has and/or in terms of existing classes that it inherits from, or *contains* (discussed in 'Class Inheritance' below). A call to `setClass()` names a new class and uses the corresponding arguments to define it.

For example, suppose we want a class of objects to represent a collection of positions, perhaps from GPS readings. A natural way to think of these in R would have vectors of numeric values for latitude, longitude and altitude. A class with three corresponding slots could be defined by:

```
Pos <- setClass("Pos", slots = c(latitude = "numeric",          longitude = "numeric", altitude =
```

The value returned is a function, typically assigned as here with the name of the class. Calling this function returns an object from the class; its arguments are named with the slot names. If a function in the class had read the corresponding data, perhaps from a CSV file or from a data base, it could return an object from the class by:

```
Pos(latitude = x, longitude = y, altitude = z)
```

The slots are accessed by the `@` operator; for example, if g is an object from the class, `g@latitude`.

In addition to returning a generator function the call to `setClass()` assigns a definition of the class in a special metadata object in the package's namespace. When the package is loaded into an R session, the class definition is added to a table of known classes.

To make the class and the generating function publicly available, the package should include `POS` in `exportClasses()` and `export()` directives in its `NAMESPACE` file:

```
exportClasses(Pos); export(Pos)
```

Defining Methods

Defining methods for an R function makes that function *generic*. Instead of a call to the function always being carried out by the same method, there will be several alternatives. These are selected by matching the classes of the arguments in the call to a table in the generic function, indexed by classes for one or more formal arguments to the function, known as the *signatures* for the methods.

A method definition then specifies three things: the name of the function, the signature and the method definition itself. The definition must be a function with the same formal arguments as the generic.

For example, a method to make a plot of an object from class `"Pos"` could be defined by:

```
setMethod("plot", c("Pos", "missing"), function(x, y, ...) {   plotPos(x, y) })
```

This method will match a call to `plot()` if the first argument is from class `"Pos"` or a subclass of that. The second argument must be missing; only a missing argument matches that class in the signature. Any object will match class `"ANY"` in the corresponding position of the signature.

Class Inheritance

A class may inherit all the slots and methods of one or more existing classes by specifying the names of the inherited classes in the `contains =` argument to `setClass()`.

To define a class that extends class "Pos" to a class "GPS" with a slot for the observation times:

```
GPS <- setClass("GPS", slots = c(time = "POSIXt"), contains = "Pos")
```

The inherited classes may be S4 classes, S3 classes or basic data types. S3 classes need to be identified as such by a call to setOldClass(); most S3 classes in the base package and many in the other built-in packages are already declared, as is "POSIXt". If it had not been, the application package should contain:

```
setOldClass("POSIXt")
```

Inheriting from one of the R types is special. Objects from the new class will have the same type. A class Currency that contains numeric data plus a slot "unit" would be created by

```
Currency <- setClass("Currency", slots = c(unit = "character"), contains = "numeric")
```

Objects created from this class will have type "numeric" and inherit all the builtin arithmetic and other computations for that type. Classes can only inherit from at most one such type; if the class does not inherit from a type, objects from the class will have type "S4".

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

`is` オブジェクトはあるクラスからのものか?

Description

オブジェクトとクラス (is) の間，または二つのクラス (extends)間の継承関係をテストし，そうした関係を確立する(setIs関数で， setClass) に対する contains= 引数への明示的な別法である.

Usage

```
is(object, class2)

extends(class1, class2, maybe = TRUE, fullInfo = FALSE)

setIs(class1, class2, test=NULL, coerce=NULL, replace=NULL,
      by = character(), where = topenv(parent.frame()), classDef =,
      extensionObject = NULL, doComplete = TRUE)
```

Arguments

object	任意の R オブジェクト.
class1, class2	その間で is 関係の定義が調べられるクラスの名前，または (より効率的に)クラスに対するクラス定義オブジェクト.
maybe, fullInfo	extends の呼び出し中で，maybe はもし関係が条件付きならば返される値. class2 が欠損した呼び出し中では， fullInfo はフラグで，もし TRUE ならば，単にクラス名だけではなく，クラス classExtension のオブジェクトのリストが返されるようにする.

coerce, replace
: In a call to setIs の呼び出し中で，オブジェクトを class2 に強制変換し，オブジェクトを is(object, class2) が value に一致するように変換するためにオプションで提供される. 下の詳細節を見よ.

test
: setIs の呼び出し中で，この関数を使うことで条件付き 関係が定義される．条件付き関係は推奨出来ずメソッド選択中に含められない．下の詳細説を見よ.

残りの引数は内部的使用且つ/または普通省略される.

extensionObject
: test, coerce, replace, by 引数の代替物；関係を記述するクラス SClassExtension からのオブジェクト． (内部的呼び出しで使われる．)

doComplete
: TRUE の時，クラス定義は間接的な関係でも増補される (内部的呼び出しで使われる．)

by
: setIs の呼び出し中で，中間のクラスの名前．強制変換は先ずこのクラスに強制変換されそこから目標のクラスに変換される． (中間の強制変換は適正でなければならない．)

where
: setIs への呼び出し中で，関係を定義するメタデータをどこに保存するか．既定はセッションのトップレベルに対しては大局的環境かそこで評価されたソースファイル．呼び出しがパッケージのソース中のファイルのトップレベル中で起きた時は，既定値はパッケージの名前空間または環境．他の使用はトリッキーであり，自分が何をしているのか真に理解していない限り普通良いアイデアではない.

classDef
: class に対するオプションのクラス定義で， setClass の呼び出しによるクラスの初期定義の過程で setIs が呼びだされる時に内部的に必要とされる．何のためにそれをするのか真に理解していない限りこの引数を 使わないこと.

関数の要約

is: 二つの引数を用い，object が class2 からのものとして扱えるかをテストする.

引数が一つならこのオブジェクトのクラスの全てのスーパークラスを返す.

extends: 最初のクラスは二番目のクラスを拡張するか? 呼び出しはもし拡張がテストを含めば maybe を返す.

引数一つで呼び出されると，値は class1 のスーパークラスのベクトルである．もし引数 fullInfo が TRUE ならば，呼び出しはクラス SClassExtension のオブジェクトの名前付きリストを返す；さもなければ単にスーパークラスの名前.

setIs: class1 を class2 の拡張(サブクラス)として定義する．もし class2 が既存の合併クラスのような仮想的クラスならば，もし含意される継承メソッドが class1 に対して動作するならば呼び出しには二つの引数だけが提供される必要がある．下の詳細節を見よ.

別法として，スーパークラスに強制変換しスーパークラスに対応する部分を置き換えるメソッドを定義する引数 coerce と replace が提供されなければならない．下の詳細節と他の節で議論されているように，この形式はその別法である setAs への対応する呼出しほどは勧められない.

引数 test は条件付き継承を許し，そこでは is() の結果はクラス定義で決定されるのではなく各オブジェクトに対してテストされる．この形式はそれが避けられる時は勧められない；特に条件付き継承は選択適用に対するメソッドを選ぶのに使われないことを注意する.

詳細

別のクラスを継承するようにあるクラスを手配することはプログラミングの鍵となる道具である． R では三つの基本的なテクニックがある，最初の二つは “単純”な継承と呼ばれるものを提供し，好まれる形式である：

1. setClass の呼び出し中の contains= 引数により．これは最も普通の機構でありまたそうあるべきである．これは新しいクラスが既存クラスの全ての構造を含み，そして特に同じクラスが指定された全てのスロットを持つように按配する．結果のクラス拡張は **単純**であるように定義され，メソッド定義に対して重要な含意を持つ (下のこのトピックに関する節を見よ).
2. class1 を setClassUnion への呼び出しでサブクラスを新しい合併クラスのメンバーにするか，または setIs の呼び出しで既存の合併クラスのクラスか既存の仮想的クラスの新しいサブクラスとしてを付け加えることで，仮想クラスのサブクラスにする．どちらのケースでも，合併クラスか他のスーパークラスに対して定義されたメソッドがサブクラスに対して正しく動作することが意図されている．これはサブクラスの構造中のある類似性に依存するかもしれず，単にスーパークラスメソッドが全てのサブクラスに適用される総称的関数の言葉で定義されるかもしれない．これらの関係はまた一般に単純である．
3. coerce と replace 引数を setAs に与える． R は任意の継承関係を認め， setAs への呼び出しによる強制変換機構の定義と同じ機構を使う．両者の違いは単に setAs は変換が起きるためには as の呼び出しが必要なのに対して， setIs の呼び出しの後にはオブジェクトは自動的にスーパークラスに変換されることである．

   自動特性は危険な部分で，主な理由は結果が可能性として動作しないメソッドを継承するサブクラスになることである．下の継承に関する節を見よ．もし含まれる二つのクラスが実際には大量のメソッドのコレクションを継承しなければ，危険は相対的に僅かかもしれない．

   スーパークラスがサブクラスが既定か離れている継承メソッドだけを継承していれば，問題はより起こりがちである．この場合，一般的な助言は強い反対意見がない限り代わりに setAs 機構を使うことである．さもなければ継承されたメソッドのどれかを書き換える用意をせよ．

この注意を与えて，この節の残りは coerce= と replace= 引数が setIs に与えられた時何が起こるかを解説する．

coerce と replace 引数は class1 オブジェクトを class2 をどのように強制変換するか，そしてどのように class2 に対応するサブクラスオブジェクトをの部分を置き換えるかを定義する関数である．これらの最初は from であるべき引数が一つの関数で，二番目は二つの引数 (from, value) の関数である．詳細については下の強制変換に関する節を見よ．

by が使用された時，強制変換のプロセスは先ずこのクラスにそれから class2 に変換される． by 引数を直接使うことはありそうもないが，クラスに関するキャッシュされた情報を定義するのに使われる．

setIs により(不可視で)返される値は class1 の改訂されたクラス定義である．

強制変換，置き換え，そしてテスト関数

coerce 引数は class1 オブジェクトを class2 オブジェクトに変える関数である．replace 引数は class1 オブジェクト (最初の引数) を， class2 (value として与えられる，第二引数)に対応するその部分を置き換えるために修正する引数が二つの関数である．換言すれば，それは表現式 as(object, class2) <- value を実装する置き換えメソッドとして作用する．

coerce と replace 関数を考える最も簡単な方法は class1 が第二のクラスのスロットを含むことで普通の意味で class2 を含む場合を考えることである．(繰り返すと，この状況では setIs を呼び出さないが，類似からそれをする時何が起きるかを示す．)

この場合 coerce 関数は単に class2 オブジェクトを，対応する class1 オブジェクトから対応するスロットを取り出すことで作る．replace 関数は class1 オブジェクト中で class2 に対応するスロットを置き換え，そして修正されたオブジェクトをその値として返す．

これらの関数の追加の議論については， setAs 関数のドキュメントを見よ．(不幸にもその関数への def 引数はここでの coerce に対応する．)

もし関数が test 引数として与えられると，継承関係はまた条件付きであり得る．これは与えられたオブジェクトが関係 is(object, class2) を満足するかどうかで TRUE か FALSE を返す引数一つの関数であるべきである．条件付き関係は，それらが正当性を決定するためにオブジェクト毎の計算を必要とするために，一般的には推奨できない．それらは普通の関係程は効率的に適用できず，そしてそれらを使うコードをより解釈困難にする傾向がある． 注意：条件付き継承はメソッドの選択適用には使われない． 条件付きスーパークラスに対するメソッドは継承されない．代わりに，条件付き関係をテストするサブクラスに対するメソッドが定義されるべきである．

継承メソッド

特定のシグネチャ (関数の一つまたはそれ以上の形式的引数にマッチするクラス) に対して書かれたメソッドは自然に引数に対応するオブジェクトは対応するクラス由来と扱うことが出来る．オブジェクトはクラスに対する全てのスロットと利用可能なメソッドを持つ．

メソッドを選択し適用するコードはこの仮定が正しいことを保証する．もし継承が “単純”ならば，つまり setClass の呼び出し中の一つまたはそれ以上の codecontains= 引数の使用で定義されるならば，余分の作業は一般的に不要である．クラスは同じ定義を使いスーパークラスから継承される．

継承が setIs の一般的な呼び出しで定義される時は，余分の計算が必要になる．この形式の継承はサブクラスがスーパークラスのスロットを単に含ま無いことを意味するが，代わりに強制変換と/または置き換えメソッドの明示的な呼び出しを必要とする．正確な計算を保証するためには，継承メソッドはメソッドの本体が評価される前に as の呼び出しを補う．

この場合に生成された as の呼び出しは引数 strict = FALSE を持ち，それが全ての適当なスロットを持つ限り，余分の情報を変換されたオブジェクト中に残すことが出来ることを意味する． (単純なサブクラスオブジェクトが変更なしに使われることを許すのはこのオプションである．) 強制変換メソッドを書くときには，このオプションの長所を利用したくなるかもしれない．

単純でない拡張を通じて継承されたメソッドは曖昧さや予期せぬ選択に導くことがあり得る．もし class2 が少数の適用可能なメソッドだけを持つ特殊化されたクラスならば，継承関係の作成は class1 の挙動にほとんど影響を与えないかもしれない．しかしもし class2 が多くのメソッドを持つクラスならば， class1 に対してある望ましくないメソッドを継承し，ある場合には期待されるメソッドを継承することに失敗することに気づくかもしれない．下の二つ目の例では，クラス "factor" からの単純でない継承はそのクラスにより S3 メソッドを継承するとされるかもしれない．しかし S3 クラスは曖昧であり，実際 "factor" ではなく "character" である．

ある種の総称的関数に対しては，単純でない拡張により継承されたメソッドは不正であるかまたは十分不正の可能性があり，従って総称関数はそうした継承を除外するように定義されてきている．例えば initialize メソッドは目標クラスのオブジェクトを返さなければならない；これは拡張が単純ならば引数オブジェクトに何の変更も加えられない

ので簡単であるが，本質的に不可能である．この理由から，総称的関数は継承に対して単純な拡張だけを強要する．この機構については setGeneric の simpleInheritanceOnly 引数を見よ．新しい総称的関数を定義する際はこの機構を使うことが出来る．

単純でない継承を認める関数に対して問題が起きれば，二つの基本的な選択がある．setIs 呼び出しから戻り setAs の呼び出しで定義される明示的な強制変換を設定する；又は好ましくない継承メソッドを上書きする class1 を含む明示的なメソッドを定義する．深刻な問題が起きている時は最初の選択がより安全である．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

inherits は S4 と非 S4 オブジェクト双方に対してほとんど常に is に同値であり少々速い．非同等性は関係の中に自明でない test= がある条件付きスーパークラスを持つクラスに適用される(普通でなく推奨できない)：これらに対して is は関係をテストするが inherits は定義から S4 オブジェクトに対する条件付き継承を無視する．

selectSuperClasses(cl) は extends(cl) と類似のセマンティックスを持つ．

Examples

```
## coerce= と replace= 引数を持つ setIs() の二つの例
## 最初の例はどのクラスも新しい継承により区別されるべき多くの
## 継承メソッドを持たぬため，かなり上手く動作する．

## 二番目の例は新しいスーパークラス "factor" がそうなるべきではない
## メソッドを継承させるため上手く動作しない．

## 最初の例：
## クラス定義(クラス "track" に対する \link{setClass} を見よ)
setClass("trackCurve", contains = "track",
         representation( smooth = "numeric"))
## "trackCurve" に類似のクラスであるが，
## "y" と "smooth" スロットに行列を許す異なった構造を持つ
setClass("trackMultiCurve",
         representation(x="numeric", y="matrix", smooth="matrix"),
         prototype = structure(list(), x=numeric(), y=matrix(0,0,0),

                               smooth= matrix(0,0,0)))
## y と smooth スロットを一列の行列にすることで
## クラス "trackCurve" からのオブジェクトを自動的に "trackMultiCurve" に変換する
setIs("trackCurve",
      "trackMultiCurve",
      coerce = function(obj) {
        new("trackMultiCurve",
            x = obj@x,
            y = as.matrix(obj@y),
            smooth = as.matrix(obj@smooth))
      },
      replace = function(obj, value) {
```

```
        obj@y <- as.matrix(value@y)
        obj@x <- value@x
        obj@smooth <- as.matrix(value@smooth)
        obj})



## 第二の例：
## "character" へのスロットを追加するクラス
setClass("stringsDated", contains = "character",
         representation(stamp="POSIXt"))

## 明示的な強制変換で因子に自動的に強制変換
setIs("stringsDated", "factor",
      coerce = function(from) factor(from@.Data),
      replace= function(from, value) {
                  from@.Data <- as.character(value); from })

ll <- sample(letters, 10, replace = TRUE)
ld <- new("stringsDated", ll, stamp = Sys.time())

levels(as(ld, "factor"))
levels(ld) # NULL になるはず--上の継承に対する節のコメントを見よ.

## 対照的に, "factor" を単純に拡張するクラスは
## そうした曖昧さを持たない
setClass("factorDated", contains = "factor",
         representation(stamp="POSIXt"))
fd <- new("factorDated", factor(ll), stamp = Sys.time())
identical(levels(fd), levels(as(fd, "factor")))
```

---

isSealedMethod　　封印されたメソッドかクラスをチェック

Description

これらの関数はメソッドかクラスがそれが定義された時に封印されているか，そして従って再定義出来ないかをチェックする.

Usage

```
isSealedMethod(f, signature, fdef, where)
isSealedClass(Class, where)
```

Arguments

f	総称的関数の引用化名.
signature	メソッドのシグネチャ中のクラス名， setMethod に提供されるようなもの.
fdef	オプションで普通省略される： f に対する総称的関数の定義.
Class	クラスの引用化名.

where メソッドやクラス名をどこで探すか．既定では，呼び出し isSealedMethod か isSealedClass のトップ環境から探す．典型的には大局的環境か関数の一つの呼び出しを含むパッケージの名前空間．

Details

R のクラスやメソッドの実装では，クラスやメソッドの定義を封印できる．基本クラス(数値と他のベクトルタイプ，行列，そして配列データ) は封印されている．これらのデータタイプに関するプリミティブ関数に対するメソッドも同様である．その効果はこれらの基本データタイプや計算の意味を再定義できないということである．より正確には，一つのデータタイプだけに依存するプリミティブ関数に対しては，基本クラスに対するメソッドは指定できない．二つの引数に依存する関数(算術演算のような)に対しては，もしこれらの引数のどちらかが基本クラスであるが共にそうではない時はメソッドを指定できる．

プログラマは sealed 引数を用いて他のクラスやメソッド定義を封印できる．

Value

関数はもしメソッドやクラスが封印されていなければ FALSE を返す (それが定義されていない場合を含む)；もしされていれば TRUE.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンについて．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンについて．)

Examples

```
## これらは共に TRUE
isSealedMethod("+", c("numeric", "character"))
isSealedClass("matrix")

setClass("track",
           representation(x="numeric", y="numeric"))
## しかしこれは FALSE
isSealedClass("track")
## そしてこれも
isSealedClass("A Name for an undefined Class")
## そしてこれらも，二つの引数のひとつだけが基本的なため
isSealedMethod("+", c("track", "numeric"))
isSealedMethod("+", c("numeric", "track"))
```

---

language-class 未評価の言語オブジェクトを表現するクラス

Description

仮想クラス "language" とそれを拡張する特定のクラスはパーサや quote の様な関数により作られる未評価のオブジェクトを表現する．

Usage

```
### これらのクラスの各々はS言語の未評価オブジェクトに対応する.
### クラス名はメソッドのシグネチャ中と幾つかの他のコンテクスト
### (as() への呼び出しのような)に登場できる.

"("
"<-"
"call"
"for"
"if"
"repeat"
"while"
"name"
"{"

### 上のクラスの各々は仮想クラス "language" を拡張する
```

クラスからのオブジェクト

"language" は仮想クラスである；それからは如何なるオブジェクトも作ることは出来ない.

他のクラスからのオブジェクトは new(Class, ...) への呼び出しで生成できる，ここで Class は引用化されたクラス名で，そして ... 引数は空かこのクラス(拡張)からの単一のオブジェクトである.

メソッド

coerce signature(from = "ANY", to = "call"). as(object, "call") に対するメソッドが存在し， as.call() を呼び出す.

LinearMethodsList-class

"LinearMethodsList" クラス

Description

要約情報を生成するための '線形化'されたメソッドリストのバージョン. メソッドの選択適用に対して使われるクラス "MethodsList" からの実際のオブジェクトは含まれる引数に関して再帰的に定義される.

クラスからのオブジェクト

関数 linearizeMlist は通常のメソッドリストオブジェクトを線形化された形式に変換する.

スロット

methods: クラス "list" のオブジェクトで，メソッドの定義.

arguments: クラス "list" のオブジェクトで，対応する形式的引数，つまり関係するメソッドテーブル中でアクティブな限りの総称的関数のシグネチャ中の引数.

classes: クラス "list" のオブジェクトで，シグネチャ中の対応するクラス．

generic: クラス "genericFunction" のオブジェクト；メソッドがそれに対応する総称的関数．

将来についての注意

linearizeMlist の現在のバージョンはクラス MethodDefinition の特徴を利用しておらず，従ってそうあるべきよりは少ない努力により多くの作業を行っている．特に，保管されたシグネチャを利用するためには関数とクラスの双方を再定義しなければならないかもしれない．現在の全ての情報は将来も得られるであろうが，現在の形式に正確に依存するコードを書かないようにしよう．

See Also

計算に関しては linearizeMlist，そしてオリジナルの再帰的な形式に対してはクラス MethodsList．

LocalReferenceClasses 参照クラスに基づく局地化オブジェクト

Description

局所参照クラスはオブジェクトを局所フレームに孤立させる修正された ReferenceClasses である．従ってそれらは変更を呼び出し環境に伝播しかえさ無い．同時に，それらは参照欄のセマンティックスを局所的に使い，標準的な R オブジェクトに適用される自動的な複製を避ける．

現在の実装は特別な構成を持たない．局所参照クラスを作るため，setRefClass() を "localRefClass" を含む contains= 引数と共に呼び出す．下の例を見よ．

局所参照クラスは本質的に R の正規の機能的なクラスに対するように操作する；つまり，変更は付値で行われ局所的なフレーム中で起こる．本質的な違いは置き換え操作(例中の twiddle 欄への変更の様に)は，形式的クラスに対してや，属性を持つデータに対してや又はそして名前付きリスト中の様に，オブジェクト全体の複製を起こさない．目的は他の欄への可能性として頻繁な変更と共には変更されないある種の欄中の大きなオブジェクトを認めることにあるが，大きな欄のコピーはしない．

Usage

```
setRefClass(Class, fields = , contains = c("localRefClass",....),
     methods =, where =, ...)
```

Details

オブジェクトの局所化は現在の実装では部分的にしか自動化されていない． $<- 演算子を使った置き換え表現式は安全である．

しかしながら，クラスの参照メソッド自体が例えば <<- を使って欄を修正すると，任意のそうしたメソッドが呼び出される前にオブジェクトが関連するフレームに対して局所的であることを確認する必要がある．さもなければ標準的な参照クラス挙動が依然として優勢になる．

局所性を保証する二つの方法がある．直接的な方法は特殊なメソッド x$ensureLocal() をオブジェクトに対して起動することである．他の方法は欄を x$field <- ... により明

示的に修正することである．参照に対する局所性を与える安直なコピーを惹き起こすためには，各オブジェクトに対してこれらの一つまたは他が一回使われるだけで十分である．下の例では，両方の機構を示す．

どのようになされるにせよ，局所化は任意のメソッドが変更を行う前に起きなければならない．(任意の環境下で成功を保証するのは困難かもしれないが，結局はコードツールの利用は少なくとも大部分このプロセスを自動化すべきである．)

Author(s)

John Chambers

Examples

```
## クラス "myIter" は実(ビッグ)データに対する BigData 欄を持ち
## そしてそれが弄る(ある理由から)ある種のパラメータに対する "twiddle" 欄を持つ

myIter <- setRefClass("myIter", contains = "localRefClass",
  fields = list(BigData = "numeric", twiddle = "numeric"))

tw <- rnorm(3)
x1 <- myIter(BigData = rnorm(1000), twiddle = tw) # OK, not REALLY big

twiddler <- function(x, n) {
  x$ensureLocal() # 詳細を見よ．この例では本当は不要
  for(i in seq(length = n)) {
      x$twiddle <- x$twiddle + rnorm(length(x$twiddle))
      ## そして何かをする ....
      ## gdb で詮索する等により x$BigData がコピーされていないことが分かる
  }
  return(x)
}

x2 <- twiddler(x1, 10)

stopifnot(identical(x1$twiddle, tw), !identical(x1$twiddle, x2$twiddle))
```

---

makeClassRepresentation
クラス定義を作る

Description

特定のクラスを記述するクラス classRepresentation のオブジェクトを作る．主としてユーティリティ関数であるが， setClass がそうするように，それに付値すること無しにクラス定義を作ることが出来る．

Usage

```
makeClassRepresentation(name, slots=list(), superClasses=character(),
                        prototype=NULL, package, validity, access,
                        version, sealed, virtual=NA, where)
```

Arguments

name	クラスに対する文字列名
slots	setClass に提供されるようなスロットクラスに対する名前付きリストであるが，もしあっても superClasses に対する名前無しの引数は除く．
superClasses	そのクラスをこのクラスは拡張するか
prototype	クラスに対する既定のデータを提供するオブジェクト，例えば prototype の呼び出しの結果．
package	クラスが保存されるパッケージに対する文字列名； getPackageName を見よ．
validity	オプションの検証メソッド． validObject と参考文献中の検証メソッドの議論を見よ．
access	アクセス情報．現在未使用．
version	バージョンコントロール用のオプションのバージョン．現在生成されるが使われない．
sealed	このクラスは封印されるか? setClass を見よ．
virtual	これは仮想的クラスであることが分かっているか?
where	クラス定義をそこから探す環境 (例えば，スロットやスーパークラス)．GenericFunctions の下のこの引数に対する議論を見よ．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

setClass

---

method.skeleton 新しいメソッドに対する骨格ファイルを作る

Description

この関数は与えられた総称的関数とシグネチャに対するメソッドを定義する setMethod の呼び出しを含むソースファイルを書く．既定ではメソッド定義は呼び出し中に埋め込まれているが，外部(先に付値された)関数にすることも出来る．

Usage

```
method.skeleton(generic, signature, file, external = FALSE, where)
```

Arguments

generic	総称的関数の文字列名，または総称的関数自体．最初の場合，結果の setMethod 呼び出しに対してそうであるように，関数は現在総称的で無くても良い．
signature	setMethod に与えられるようなメソッドのシグネチャ．
file	出力ファイルの文字列名，または買い込み可能なコネクション．既定では総称的関数名とシグネチャ中のクラスがアンダースコア文字で連結される，ファイル名は普通 ".R" で終わる． 複数のメソッド骨格を一つのファイルに書き込むには，先ずファイルコネクションを開きそれからそれを複数呼び出し中の method.skeleton() に渡す．
external	メソッドに対する関数定義がソースファイル中に付値された別個のオブジェクトであるべきか，または setMethod への呼び出し中にインラインで埋め込まれるべきかを制御するフラグ．もし文字列として与えられると，これは外部関数に対する名前として使われる；既定では名前は総称的関数とシグネチャをアンダースコア文字で連結する．
where	関数をその中で探す環境；既定では method.skeleton の呼び出しのトップレベル環境．

Value

不可視の file 引数であるが，関数はその副作用のために使われる．

See Also

setMethod, package.skeleton

Examples

```
setClass("track", representation(x ="numeric", y="numeric"))
method.skeleton("show", "track")            ## show_track.R を書き込む
method.skeleton("Ops", c("track", "track")) ## "Ops_track_track.R" を書き込む

## 複数のメソッド骨格を一つのファイルに書き込む
con <- file("./Math_track.R", "w")
method.skeleton("Math", "track", con)
method.skeleton("exp", "track", con)
method.skeleton("log", "track", con)
close(con)
```

MethodDefinition-class
メソッド定義を表現するクラス

Description

これらのクラスは関数がメソッド定義として保存され使われる基本クラス "function" を拡張する．

Details

メソッド定義オブジェクトは関数がメソッドとしてどのように使われるかの付加情報を持つ関数である．スロット target はそれに対してメソッドが選択適用されるクラスのシグネチャであり，スロット defined はメソッドが元来指定されているシグネチャ (つまり setMethod へのある呼び出し中に現れるもの)である．

クラスからのオブジェクト

setMethod への呼び出しでメソッドを設定する行動はこのクラスのオブジェクトを作る．それらを直接作ることは賢明ではない．

クラス "SealedMethodDefinition" は引数 sealed = TRUE を持つ setMethod への呼び出しで作られる．これは "MethodDefinition" と同じ表現を持つ．

スロット

.Data: クラス "function" のオブジェクト；定義のデータ部分．

target: クラス "signature" のオブジェクト；それに対してメソッドが必要とされるシグネチャ．

defined: クラス "signature" のオブジェクト；それに対してメソッドが見つけられたシグネチャ．もしメソッドが継承されていたならば，これは target と同じではない．

generic: クラス "character" のオブジェクト；それに対してメソッドが作られた関数．

拡張

データ部分から，クラス "function".
直接，クラス "PossibleMethod".
"function" から，クラス "OptionalMethods".

See Also

特定の総称的関数に関連したメソッドのセットを定義するオブジェクトについてはクラス MethodsList．これらのオブジェクト中に保管された個々のメソッド定義はクラス MethodDefinition またはその拡張に由来する． callNextMethod により使われる拡張についてはクラス MethodWithNext.

Methods　　メソッドに関する一般的情報

Description

このドキュメントはメソッドがどのように動作するか，そして **methods** パッケージが R の残りとどのように関わるかに関するある種の一般的なトピックを解説する．この情報は通常メソッドとクラスの利用を始めるためには必要ではないが，ある程度野心的なプロジェクトや，何かが期待されたように動作しない場合に役立つかもしれない．

節 “メソッドはどのように動作するか” は基礎にある機構を解説する；節 “S3 総称的関数に対するメソッド” は S4 クラスとメソッドが古い S3 メソッドと関わる時に適用される規則を与える；節 “メソッド選択と適用” はクラス定義がどのメソッドが使われるかをどのように決めるのかに関するより詳細を与える；節 “総称的関数” はオブジェクトとしての総称的関数を議論する．特にクラス定義に関する追加情報については Classes を見よ．

メソッドはどのように動作するか

総称的関数はそれに他の関数のコレクション (メソッド)を関連付け，それらは総称的関数と同じ形式的引数を持つ．総称的関数自体についてのより詳細は下の節 “総称的関数” を見よ．

各々の R パッケージはそのパッケージ中でメソッドが定義されている各総称的関数に対応するメソッドのメタデータオブジェクトを含む．パッケージが R セッション中にロードされると，各総称的関数に対応するメソッドは キャッシュされる，つまり以前にロードされたパッケージのメソッドと並んで総称的関数の環境中に保存される．このメソッドの混ぜ合わされたテーブルは，適用可能なメソッドを見つけるためにクラス継承と可能性としてグループ総称的関数を利用して総称的関数からのメソッドを適用したり選択したりするのに使われる．下の節 “メソッドの選択と適用” を見よ．キャッシング計算は各総称的関数のただ一つのバージョンが大局的に可視になることを保証する；異なった付加されたパッケージは総称的関数のコピーを含むかもしれないが，これらはメソッド選択に関しては同じ働きをする．対照的に，それらが異なった package スロットを持つなら，同じ関数名が一つ以上の総称的関数を参照することが可能である．後者の場合 R 関数は無関係と見做す．総称的関数はは名前とパッケージの組み合わせで定義される．下の節 “総称的関数” を見よ．

総称的関数に対するメソッドは，メソッドを定義する `setMethod` への呼び出し中の対応する `signature` に従って保存される．シグネチャは一つのクラス名を総称的関数への形式的引数のサブセットの各々と関連させる．どの形式的引数が利用可能か，そしてそれらが登場する順序は総称的関数自体の `"signature"` スロットにより決定される．既定では，総称的関数のシグネチャは ... を除く全ての形式的引数から成り立ち，関数定義中にそれらが登場する順序を持つ．

総称的関数のシグネチャ中の後に続く引数は，もしそのシグネチャ中にこれらの引数を含む如何なるメソッドもまだ指定されていなければ アクティブで無い．アクティブでない引数はキャッシュされたメソッドのラベリングでは不要で使われない． (違いはどのメソッドが選択適用されるかは変えないが，しかしアクティブでない引数を無視することは選択適用の効率性を向上させる．)

総称的関数のシグネチャ中の全ての引数は関数が呼び出されると，伝統的な S の遅延評価規則を使う代わりに，評価される．従ってシグネチャからシンボリックに処理される必要がない任意の引数を除外することが重要である (関数 `substitute` の第一引数のような)．既定表現ではなく実際の引数だけが評価されることを注意する．欠損引数はクラス `"missing"` としてメソッドの選択に登場する．

キャッシュされたメソッドは環境オブジェクトに保存される．付値に対して使われた名前はクラス名とメソッドのシグネチャ中のアクティブな引数を連結したものである．

S3 総称的関数に対するメソッド

S4 メソッドは S3 総称的関数の第一引数に対応する S3 メソッドも持つ関数に対して必要になるかもしれない –どちらも正規の R 関数で， S3 選択適用関数 `UseMethod` への呼び出しか，真の関数ではなく直接に C コードが使われるプリミティブ関数のセットの一つであるとする．どちらの場合も S3 メソッドの選択適用は最初の引数のクラスかプリミティブな二項演算子の一つへの呼び出し中のどちらかの引数のクラスを眺める． S3 メソッドは総称的関数と同じ引数を持つ通常の関数である (プリミティブ関数に対しては形式的引数は実際はオブジェクトの一部ではないが，プリントされたり `args()` で眺められるときにはシミュレートされる)． S3 メソッドの“シグネチャ”にはメソッドが付値される名前で特定され，総称的関数の名前に `"."` とクラス名が後に続く．詳細は S3Methods を見よ．

これらの関数のどれかに S4 クラスに対応するメソッドを実装するには二つの可能性がある； S4 メソッドとしてか S4 クラス名を持つ S3 メソッドとしてかである． S3 メソッド

は意図するシグネチャが第一引数を持ち他には何も持たない時だけ可能である．この場合推奨できるアプローチは S3 メソッドを定義しそして S4 メソッドの定義として同一の関数を提供することである．もし S3 総称的関数が `f3(x, ...)` で新しいメソッドに対する S4 クラスが `"myClass"` なら：

```
f3.myClass <- function(x, ...) { ..... }
setMethod("f3", "myClass", f3.myClass)
```

S3 と S4 メソッドを共に定義する理由は以下の通りである：

1. S4 メソッドそのものは S3 総称的関数が直接呼び出されるときは見えない．しかしながらプリント関数と演算子は例外である．内部 C コードはオブジェクトが S4 オブジェクトの時，そしてその時だけ S4 メソッドを探す．例の中では，クラス `"myFrame"` に対する `` `[` `` メソッドがこのクラスのオブジェクトに対して常に呼び出される．
   同じ理由から， S3 クラスに対して定義された S4 メソッドは非 S4 クラスに対する内部コードからは呼び出されない． (例中の関数 `Math` とクラス `"data.frame"` を見よ．)
2. S3 メソッドはもし 任意の適格な非既定の S4 メソッドがあれば単独では呼び出されない．(例中の関数 `f3` とクラス `"classA"` の例を見よ．)

選択の計算の詳細は下で与えられる．

S4 総称的関数でない既存関数に対して S4 メソッドが定義されると (既存の関数が S3 総称的であろうとなかろうと)， S4 総称的関数が既存関数とそれが見つかったパッケージに対して作られる (より正確には指定されたか通常の関数から推測された暗黙の関数；`implicitGeneric` を見よ)． `setMethod` への最初の呼び出しの後にメッセージがプリントされる；これはエラーではなく単に総称的関数が作られたことの覚書である．総称的関数を明示的に呼び出し

```
setGeneric("f3")
```

で作ればメッセージが避けられるが効果は同じである．既存関数は S4 総称的関数の既定メソッドになる．プリミティブな関数は同じように動作するが， S4 総称的関数は明示的には作られない (下で議論されるように)．

S4 と S3 メソッドは可能な限り一貫性のある継承規則に従うようにデザインされている．S3 クラスは S3 クラスが正しいパターンを指定する呼び出しと共に `setOldClass` への呼び出しで登録されていれば任意の S4 メソッド選択に対して使うことが出来る． S4 クラスは任意の S3 メソッド選択に対して使うことが出来る；S4 オブジェクトが検出された時，S3 メソッド選択は `extends(class(x))` の内容を S3 継承の同値物として使う (継承は最初の呼び出しの後にキャッシュされる)．

既存の S3 メソッドは S4 サブクラスに対して期待されるようには動作しないかもしれない．そうした場合 `asS3` や `S3Part` の様なユーティリティが役立つかもしれない．もし S3 メソッドが S4 オブジェクトに対して失敗すると， `asS3(x)` が代わりに渡されるかもしれない；もし S3 メソッドにより返されるオブジェクトが S4 オブジェクト中に統合される必要があると，例中のクラス `"myFrame"` に対するメソッド中のように， `S3Part` に対する置き換え関数が役立つかもしれない．

S3 と S4 メソッドの両方を定義する理由をここで説明する． S3 総称的関数に依然として直接アクセスする呼び出しはプリミティブ関数の場合を除き S4 メソッドを見ない．これは S3 総称的関数を移入するが S4 バージョンを移入しない名前空間からの総称的関数の呼び出しは S3 メソッドだけを見ることを意味する．他方で，S3 メソッドは S4 総称的関数からその既定 (`"ANY"`) メソッドの一部としてだけ選択される．もし継承された S4 の非既定メソッドがあれば，これらは 任意の S3 メソッドに優先して選ばれる．

プリミティブ関数として実装された S3 総称的関数 (二項演算子を含む)は S3 メソッドだけの認識への例外である．これらの関数は内部 C コードから S4 と S3 メソッドの双方を

選択適用する． S3 にせよ S4 にせよ明示的な総称的関数は無い．内部コードは，最初の引数または二項演算子の場合はどちらかの引数が S4 オブジェクトならば S4 メソッドを探す．もし S4 メソッドが何も見つからなければ S3 メソッドが探される．

S4 メソッドは S3 総称的関数と S3 クラスに対して定義できるが，もし関数がプリミティブならば，そうしたメソッドは問題のオブジェクトが S4 オブジェクトでなければ選択されない．例えば下の例では，関数 `f3()` に対するシグネチャ `"data.frame"` に対する S4 メソッドは S3 オブジェクト `df1` に対して呼び出される．プリミティブ関数 `` `[` `` に対する類似の S4 メソッドはそのオブジェクトに対しては無視されるが， `"data.frame"` を継承する S4 オブジェクト `mydf1` に対して呼び出される． S3 と S4 メソッドの両方を定義すればこの非一貫性は除かれる．

メソッドの選択と適用：詳細

総称的関数への呼び出しが評価される時，メソッドがシグネチャ中の実際の引数のクラスに対して選択される．最初にキャッシュされたメソッドテーブルで正確なマッチが探される；つまり各非欠損引数に対する `class(x)` の文字列値，そして欠損値に対しては `"missing"` で定義されるシグネチャの下に保管されたメソッドである．総称的関数への呼び出し中の実際の引数に対して如何なるメソッドも直接には見つからなければ，実際の引数に対するスーパークラス情報を用いて引数に対して使用できるメソッドをマッチする努力がなされる．

各クラス定義は新しいクラスの一つまたはそれ以上の スーパークラスを含むかもしれない．最も単純で最も普通の指定は `setClass` への呼び出し中の `contains=` 引数によるものである．この引数中で命名されたクラスは新しいクラスのスーパークラスである．スーパークラスを定義する二つの追加の機構が存在する． `setClassUnion` への呼び出しは合併のメンバーの各々のスーパークラスである合併クラスを作る． `setIs` への呼び出しは新しいクラス中にスーパークラス表現を含む単純なものではない継承関係を作ることが出来る．引数 `coerce` と `replace` はスーパークラスへの変換とスーパークラスに対応する部分を置き換えるメソッドを提供する． (更に `test=` 引数は条件付き継承を可能にする；条件付き継承は推奨されずメソッド選択中で使われない．) 三つ全ての機構はメソッド選択の目的にとっては同等に扱われる：それらは特定のクラスの 直接のスーパークラスを定義する．機構についてのより詳細については `Classes` を見よ．

直接のスーパークラス自体は同じ機構のいずれかで定義されたスーパークラスを持つかもしれず，更に同様に世代を持つかもしれない．これらの情報全てを一緒にしてこのクラスに対するスーパークラスの完全なリストを形作る．スーパークラスリストは R セッションの間キャッシュされるクラス定義中に含まれる．リストの各要素は関係の性質を記述する (詳細は `SClassExtension` を見よ)．要素中には関係のパス長を含む `distance` スロットが含まれる：直接のスーパークラスは 1 (それらを定義した機構とは無関係に)，そしてこれらのクラスの直接のスーパークラスは 2，云々．加えて任意のクラスは暗黙のうちにクラス `"ANY"` をスーパークラスとして持つ． `"ANY"` への距離は任意の実際のクラスへの距離よりも大きいとされる．欠損引数に対応する特殊なクラス `"missing"` は `"ANY"` だけをスーパークラスとして持つ一方， `"ANY"` はスーパークラスを持たない．

クラス定義が作られたり修正された際，スーパークラスは順序付けられ，最初は距離による全てのスーパークラスの安定なソートによる．もしスーパークラスのセットが重複していると (つまり，もしあるクラスが一つ以上の関係を通じて継承されていると)，もし可能ならばこれらは取り除かれ，スーパークラスのリストが全ての直接のスーパークラスのスーパークラスと一貫性を持つようにする．詳細は継承に冠する参考文献を見よ．

スーパークラスに関する情報はクラス定義がプリントされる時要約される．

メソッドが継承により選択される際には，アクテイブなシグネチャ内の各引数に対して，直接なクラスかそのスーパークラスの一つの組み合わせに直接対応する全てのメソッドに対するデーブル中で検索が行われる．例えば，シグネチャ内にただ一つの引数だけが

あり対応するオブジェクトのクラスが "dgeMatrix" (推奨パッケージ Matrix からの)であると仮定する．このクラスは二つの直接のスーパークラスを持ち，それらを通じて4つの追加のスーパークラスを持つ．メソッド選択はこれらのクラスの一つか "ANY" でラベルが付いた直接指定されたメソッドのテーブル中の全てのメソッドを見つける．

シグネチャ中に複数の引数があるときは，各引数は類似の継承クラスのリストを作る．可能なマッチはすると各引数からのクラスの全ての組み合わせである (全ての可能な組み合わせの配列を生成する関数 outer を思い起こそう)．検索は今やこのクラスの組み合わせのどれかにマッチする全てのメソッドを見出す．各引数に対して，その引数のスーパークラスのリスト中の位置がどのメソッドまたはメソッド群が (もし同じクラスが一度以上登場する場合)一番良くマッチするかを定義する．ただ一つの引数がある場合，最良のマッチは曖昧さがない．複数の引数の場合は全ての引数に対する最良のマッチ中にはゼロか一つのマッチがあるかもしれない．

もし最良のマッチがなければ，選択は曖昧でありどのメソッドが選ばれたかを注意するメッセージと (辞書式順序で最初のメソッド)そして他のどのようなメソッドが選択可能であったかをプリントする．曖昧さは普通ユーザが制御できることではないので，これは警告ではない．パッケージの作者は testInheritedMethods の呼び出しにより可能な曖昧な継承について自分のパッケージを吟味すべきである．

継承されたメソッドが選択された時，選択は総称的関数中にキャッシュされ，同じ呼び出しを持つ将来の呼び出しは検索を繰り返す必要はない．不正な選択に繋がる可能性があるので，キャッシュされた継承された選択はそれ自体将来の継承選択中では使われない．もし継承性の計算を再び繰り返したければ (例えば新しくロードしたパッケージがこのセッションですでに使っているよりもより直接的なメソッドを持つため)，resetGeneric を呼び出す．それらを含むクラスとメソッドは同じパッケージから由来する傾向があるので，現在の実装は新しいパッケージがロードされる度全ての総称的関数をリセットしない．

総称的関数への呼び出しを通じて開始されるのとは別に，メソッド選択は関数 selectMethod の呼び出しにより明示的に行うことが出来る．

メソッドが一旦選択されると評価器はその中でメソッドが評価される新しいコンテキストを作る．コンテキストは総称的関数への呼び出しからの引数で初期化される．これらの引数は再マッチされない．総称的関数のシグネチャ内の全ての引数は評価されている (現在アクテイブでないものを含み)；シグネチャ内に無い引数は言語の通常の遅延評価規則に従う．もし呼び出し中に引数が欠損していたら，もしあればその既定表現式は評価されていない．メソッドの選択適用はそうした引数に対して常にクラス missing を使うからである．

従って総称的関数への呼び出しは二つのコンテキストを持つ：一つは関数用で二つ目はメソッド用である．引数オブジェクトは二番目のコンテキストにコピーされるが，非標準的な総称的関数中で作られた局所的なオブジェクトはどれもされない．もう一つの重要な違いは二番目のコンテキストの (“囲み”)環境は関数としてのメソッドの環境であり，従ってそうした環境を用いた全ての R プログラミングテクニックは通常の関数としてメソッド定義に適用される．

メソッドの選択と適用のこれ以上の議論については最初の参考文献を見よ．

総称的関数

原則として総称的関数はメソッドを選択し選択されたメソッドへの呼び出しを評価する内部関数である standardGeneric() を評価する任意の関数であって良い．実際には総称的関数はクラス "function" のサブクラス由来に加えてクラス genericFunction を拡張する特殊なオブジェクトである．そうしたオブジェクトはそれらのメソッドを処理するのに必要な情報を定義するスロットを持つ．それらは又特殊化された環境を持ち，メソッド選択中で使われるテーブルを持つ．

オブジェクト中のスロット "generic" と "package" は総称的関数自体と関数が定義されているパッケージの文字列名である．クラスと同様に，R で総称的関数は二つの名前で一意的に定義される．

異なったパッケージに関連する同じ名前の総称的関数が存在しうる (そうした関数を明快に区別することは常に簡単で無いことは避けられないが)．一方で，R は現在のセッションや他のアクテイブな R のバージョン中では，特定の関数とパッケージ名の組み合わせを持つ総称的関数には唯一の定義が関連付けられることを強要する．

特定の総称的関数に対するメソッドのテーブルは，この意味で，しばしば幾つもの他のパッケージに広がっている．与えられた総称的関数に対するメソッドの全体のセットは，新しいパッケージがロードされるので，セッションの間に変わるかもしれない．各テーブルは総称的関数に対して仮定されたシグネチャ中で一貫していなければならない．

R は標準と非標準総称的関数を区別し，前者はメソッドの選択適用飲みを行う関数本体を持つ．殆どの部分に対し，区別は単に単純化のためである：総称的関数がメソッド呼び出しの選択適用だけをすることを知っていることはある種の効率化を許し，またある種の不確実さを減らす．

殆どのケースで，総称的関数は対応するパッケージ中のその名前に対応する可視的な関数である． 暗黙の総称的関数と R のプリミティブな関数の処理に必要な特殊計算という二つの例外がある．もしパッケージが関数を非総称的のままにしたいがもしそれが総称的であればそうなるであろうような制約を設けたければ，パッケージはパッケージ中の関数の暗黙の総称的関数のテーブルを含むことが出来る．そうした暗黙の総称的関数はパッケージのインストールの過程で作ることが出来，本質的に総称的関数と可能性としてそれに対するメソッドを定義しそれから関数をその非総称的形式に戻すことでなされる． (これがどのように行われるかは implicitGeneric を見よ．) この機構は主に S4 メソッドを無視したい R 中のより古いパッケージ中の関数に対して使われる．この場合でも，実際の機構はもし何か特別なことが指定されなければならない場合にだけ必要となる．全ての関数は自動的に定義される対応する暗黙の総称的バージョンを持つ (暗黙のうちにであり，暗黙の総称的関数と呼ばれる所以である)．この関数は非総称的関数と同じ引数を持つ標準的な総称的関数がであり，非総称的バージョンを既定(そして唯一の)メソッドに持ち，総称的なシグネチャは ... を除く全ての形式的引数である．

暗黙の総称的関数機構はある種の既定定義のアスペクトを上書きするためにだけ必要になる．一つの理由はシグネチャからある引数を取り除くことである．字義通りに解釈されたり，言語の遅延評価がそれに対して必要になったりする引数は総称的関数のシグネチャに含まれてはなら無い．なぜならシグネチャ中の引数はメソッドを選択するために評価されるからである．例えば関数 `with` 中の引数 `expr` は字義通りに扱われ従ってシグネチャから除外されなければならない．

もし既存の非総称的な関数が既定メソッドとして不適当なら暗黙の総称的関数を定義する必要があるかもしれない．恐らく関数はある種のオブジェクトのクラスだけに適用され，パッケージのデザイナーは一般的な既定メソッドを持たないことを好むかもしれない．別の可能性としてデザイナーはもし関数が総称的ならばある種のクラスに対して適当なメソッドに関するあるアイデアを持っているかもしれない．適度に現代的なパッケージでは，これら全てのケースに対する単純なアプローチは関数を単に総称的として定義することである．暗黙の総称的性機構は主にメソッドパッケージが利用可能であることを要求したくないより古いパッケージに対して魅力的である．

基本パッケージ中のプリミティブ関数として実装されている関数に対して明快に可視的ではない総称的関数が定義されることがある．プリミティブ関数はプリントされる時は通常の関数のように見えるが，実際は関数オブジェクトではなく R の評価器により基礎にある C コードの直接の呼び出しと解釈される二つのタイプのオブジェクトである．それらの全ての正当化は効率化であるので， R はプリミティブ関数を総称的関数オブジェクトの背後に隠蔽することを拒む．殆どのプリミティブ関数に対してメソッドを定義でき，対応するメタデータオブジェクトがそれを保存するために作られる．プリミティブ

関数の呼び出しは依然として直接に C コードに向かい，それはある場合には適用可能なメソッドをチェックする．“ある場合”の定義は，セッション中にロードされたあるパッケージに対してメソッドが検出されている必要があり，最初の引数に対して `isS4(x)` が `TRUE` (もしくは二項演算子の場合は二番目の引数に対して)であることである．メソッドが検出されているかどうかを関係する関数に対して `isGeneric` を呼び出すことで検査でき，メソッドが検出されているかどうかを `getGeneric` を呼び出すことで総称的関数を吟味することが出来る．総称的関数に関するより詳細は最初の引用文献と *R Internals* の第2節を見よ．

メソッド定義

全てのメソッド定義は `MethodDefinition` クラスからのオブジェクトとして保管される．総称的関数のクラスと同様，このクラスは通常の R 関数をある追加のスロット付きで拡張する：総称的関数の名前とパッケージ，そして二つのシグネチャスロット `"defined"` と `"target"` を持つ `"generic"` である．最初のスロットはメソッドが `setMethod` への呼び出しで作られる時提供されるシグネチャである．`"target"` スロットは継承メソッドが，上で説明されたように，選択後にキャッシュされると，コピーが適当な `"target"` シグネチャ付きで作られる．例えば `showMethods` からの出力は両方のシグネチャを含む．

効率性と一貫性の理由からメソッドの選択適用の機構は引数の再マッチをしないので，メソッド定義は総称的関数と同じ形式的引数を持つことが要求される．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (For the R version: see section 10.6 for method selection and section 10.5 for generic functions).

Chambers, John M.(2009) *Developments in Class Inheritance and Method Selection* https://statweb.stanford.edu/~jmc4/classInheritance.pdf.

Chambers, John M. (1998) *Programming with Data* Springer (For the original S4 version.)

See Also

より個別の情報には `setGeneric`, `setMethod` そして `setClass` を見よ．

メソッド中の ... の使用に対しては dotsMethods を見よ．

Examples

```
## 登録された S3 クラスを拡張するクラスが
## そのクラスの S3 メソッドを継承する

setClass("myFrame", contains = "data.frame",
    representation(timestamps = "POSIXt"))

df1 <- data.frame(x = 1:10, y = rnorm(10), z = sample(letters,10))

mydf1 <- new("myFrame", df1, timestamps = Sys.time())

## "myFrame" objects inherit "data.frame" S3 methods; e.g., for `[`

mydf1[1:2, ] # データフレームオブジェクト(追加属性を持つ)

## "myFrame" クラスに固有のメソッド


setMethod("[",
```

```
    signature(x = "myFrame"),
    function (x, i, j, ..., drop = TRUE)
    {
        S3Part(x) <- callNextMethod()
        x@timestamps <- c(Sys.time(), as.POSIXct(x@timestamps))
        x
    }
)

mydf1[1:2, ]


setClass("myDateTime", contains = "POSIXt")

now <- Sys.time() # class(now) は c("POSIXct", "POSIXt")
nowLt <- as.POSIXlt(now)# class(nowLt) is c("POSIXlt", "POSIXt")

mCt <- new("myDateTime", now)
mLt <- new("myDateTime", nowLt)

## S4 オブジェクトに対する S3 メソッドは S4 継承を用いて選択される
## オブジェクト mCt と mLt は異なった S3Class() 値を持つが，これは使われない
f3 <- function(x)UseMethod("f3") # 継承を説明するための S3 総称的関数

f3.POSIXct <- function(x) "The POSIXct result"
f3.POSIXlt <- function(x) "The POSIXlt result"
f3.POSIXt <- function(x) "The POSIXt result"

stopifnot(identical(f3(mCt), f3.POSIXt(mCt)))
stopifnot(identical(f3(mLt), f3.POSIXt(mLt)))



## S4 オブジェクトはその S4 "継承"に従い S3 メソッドを選択する


setClass("classA", contains = "numeric",
         representation(realData = "numeric"))

Math.classA <- function(x) {(getFunction(.Generic))(x@realData)}
setMethod("Math", "classA", Math.classA)


x <- new("classA", log(1:10), realData = 1:10)

stopifnot(identical(abs(x), 1:10))

setClass("classB", contains = "classA")

y <- new("classB", x)

stopifnot(identical(abs(y), abs(x))) # (バージョン 2.9.0 またはそれ以前はここで失敗する)

## S3 総称的関数：単に説明目的のため
f3 <- function(x, ...) UseMethod("f3")

f3.default <- function(x, ...) "Default f3"
```

```
## classA に対する S3 メソッド(のみ)
f3.classA <- function(x, ...) "Class classA for f3"

## 数値に対する S3 と S4 メソッド
f3.numeric <- function(x, ...) "Class numeric for f3"
setMethod("f3", "numeric", f3.numeric)

## classA に対する S3 メソッドと
## classB に対する最近接の継承 S3 メソッドが見つからない

f3(x); f3(y) # どちらも "numeric" メソッドを選択する

## 自然な継承を得るために，同一の S3 と S4 メソッドを定義する
setMethod("f3", "classA", f3.classA)

f3(x); f3(y) # 今や両者は "classA" メソッドを選択する

## S3 オブジェクトに対しては，またはもし S4 総称的関数を含まない
## パッケージから呼びだされるときは S4 と並んで S3 メソッドを定義する必要がある

MathFun <- function(x) { # Math グループに対するよりスマートな "data.frame" メソッド
  for (i in seq(length = ncol(x))[sapply(x, is.numeric)])
    x[, i] <- (getFunction(.Generic))(x[, i])
  x
}
setMethod("Math", "data.frame", MathFun)

## S4 メソッドはデータフレームを含む S4 クラスに対して動作するが，
## データフレームオブジェクトに対してはそうでは無い(S4 オブジェクトではない)

try(logIris <- log(iris)) # 古いメソッドからのエラーを受け取る

## 同じ計算をする S3 メソッドを定義する

Math.data.frame <- MathFun

logIris <- log(iris)
```

MethodsList-class　　クラス *MethodsList*，廃止されたメソッド表現

Description

このオブジェクトのクラスはメソッドの選択適用を制御するためにパッケージのオリジナルの実装で使われた．その使用は今は廃止されているが，オブジェクトは総称的関数の既定メソッドスロットとして登場する．これと他の残りの使用は将来取り除かれるであろう．

現代的な代替物については listOfMethods を見よ.

このドキュメント中の詳細は旧式のオブジェクトの解析を許すために残されている.

Details

関数 f が形式的引数 x と y を持つとする. その関数に対するメソッドリストオブジェクトはその argument スロットとしてオブジェクト as.name("x") を持つ. もし x に対応する実際の引数がクラス "track" であれば "track" という名前のメソッドの要素が選択される. もしそうした要素があれば, それは一般に関数化別のメソッドリストオブジェクトであり得る.

最初のケースでは, 関数は x がクラス "track" である任意の呼び出しについて使われるメソッドを定義する. ふたつ目のケースでは, 新しいメソッドリストオブジェクトは残りの形式的引数に依存する利用可能なメソッドを定義する.

各メソッドは概念的には シグネチャに対応する ; これはクラスの名前付きのリストで, 名前はあるもしくは全ての形式的引数に対応する. 最初の例では, もし x に対してクラス "track" を選択し, 選択が別のメソッドリストであることがわかり, そして y に対してクラス "numeric" を選択すればシグネチャ x = "track", y = "numeric" に関連するメソッドが作られる.

スロット

argument: クラス "name" のオブジェクト. このレベルでの選択適用のために使われる引数の名前.

methods: この引数に対して 明示的に定義されたメソッド(そしてメソッドのリスト) の名前付きリスト. 名前はクラス名で, そして対応する要素は対応する引数がそのクラスをを持てば使われるメソッドかメソッドを定義する. 詳細は下を見よ.

allMethods: 名前付きリストで, methods スロットからの全ての直接的に定義されたメソッドに任意の継承メソッドを含む. メソッドテーブルが選択適用に対して使われるときは無視される (Methods を見よ).

拡張

直接なクラス "OptionalMethods".

Methods_Details *General Information on Methods*

Description

This documentation covers some general topics on how methods work and how the **methods** package interacts with the rest of R. The information is usually not needed to get started with methods and classes, but may be helpful for moderately ambitious projects, or when something doesn't work as expected.

For additional information see documentation for the important steps: (setMethod(), setClass() and setGeneric()). Also Methods_for_Nongenerics on defining formal methods for functions that are not currently generic functions; Methods_for_S3 for the relation to S3 classes and methods; Classes_Details for class definitions and Chapters 9 and 10 of the reference.

How Methods Work

A call to a generic function selects a method matching the actual arguments in the call. The body of the method is evaluated in the frame of the call to the generic function. A generic function is identified by its name and by the package to which it correspond. Unlike ordinary functions, the generic has a slot that specifies its package.

In an R session, there is one version of each such generic, regardless of where the call to that generic originated, and the generic function has a table of all the methods currently available for it; that is, all the methods in packages currently loaded into the session.

Methods are frequently defined for functions that are non-generic in their original package,. for example, for function `plot()` in package **graphics**. An identical version of the corresponding generic function may exist in several packages. All methods will be dispatched consistently from the R session.

Each R package with a call to `setMethod` in its source code will include a methods metadata object for that generic. When the package is loaded into an R session, the methods for each generic function are *cached*, that is, added to the environment of the generic function. This merged table of methods is used to dispatch or select methods from the generic, using class inheritance and possibly group generic functions (see `GroupGenericFunctions`) to find an applicable method. See the "Method Selection and Dispatch" section below. The caching computations ensure that only one version of each generic function is visible globally; although different attached packages may contain a copy of the generic function, these behave identically with respect to method selection.

In contrast, it is possible for the same function name to refer to more than one generic function, when these have different `package` slots. In the latter case, R considers the functions unrelated: A generic function is defined by the combination of name and package. See the "Generic Functions" section below.

The methods for a generic are stored according to the corresponding `signature` in the call to `setMethod` that defined the method. The signature associates one class name with each of a subset of the formal arguments to the generic function. Which formal arguments are available, and the order in which they appear, are determined by the `"signature"` slot of the generic function itself. By default, the signature of the generic consists of all the formal arguments except ..., in the order they appear in the function definition.

Trailing arguments in the signature of the generic will be *inactive* if no method has yet been specified that included those arguments in its signature. Inactive arguments are not needed or used in labeling the cached methods. (The distinction does not change which methods are dispatched, but ignoring inactive arguments improves the efficiency of dispatch.)

All arguments in the signature of the generic function will be evaluated when the function is called, rather than using lazy evaluation. Therefore, it's important to *exclude* from the signature any arguments that need to be dealt with symbolically (such as the `expr` argument to function `with`). Note that only actual arguments are evaluated, not default expressions. A missing argument enters into the method selection as class `"missing"`.

The cached methods are stored in an environment object. The names used for assignment are a concatenation of the class names for the active arguments in the method signature.

Method Selection: Details

When a call to a generic function is evaluated, a method is selected corresponding to the classes of the actual arguments in the signature. First, the cached methods table is searched for an exact match; that is, a method stored under the signature defined by the string value of `class(x)` for each non-missing argument, and `"missing"` for each missing argument. If no method is found directly for the actual arguments in a call to a generic function, an attempt is made to match the available methods to the arguments by using the superclass information about the actual classes. A method

found by this search is cached in the generic function so that future calls with the same argument classes will not require repeating the search. In any likely application, the search for inherited methods will be a negligible overhead.

Each class definition may include a list of one or more direct *superclasses* of the new class. The simplest and most common specification is by the `contains=` argument in the call to `setClass`. Each class named in this argument is a superclass of the new class. A class will also have as a direct superclass any class union to which it is a member. Class unions are created by a call to `setClassUnion`. Additional members can be added to the union by a simple call to `setIs`. Superclasses specified by either mechanism are the *direct* superclasses.

Inheritance specified in either of these forms is *simple* in the sense that all the information needed for the superclass is asserted to be directly available from the object. R inherited from S a more general form of inheritance in which inheritance may require some transformation or be conditional on a test. This more general form has not proved to be useful in general practical situations. Since it also adds some computational costs non-simple inheritance is not recommended. See `setIs` for the general version.

The direct superclasses themselves may have direct superclasses and similarly through further generations. Putting all this information together produces the full list of superclasses for this class. The superclass list is included in the definition of the class that is cached during the R session. The *distance* between the two classes is defined to be the number of generations: `1` for direct superclasses (regardless of which mechanism defined them), then `2` for the direct superclasses of those classes, and so on. To see all the superclasses, with their distance, print the class definition by calling `getClass`. In addition, any class implicitly has class `"ANY"` as a superclass. The distance to `"ANY"` is treated as larger than the distance to any actual class. The special class `"missing"` corresponding to missing arguments has only `"ANY"` as a superclass, while `"ANY"` has no superclasses.

When a method is to be selected by inheritance, a search is made in the table for all methods corresponding to a combination of either the direct class or one of its superclasses, for each argument in the active signature. For an example, suppose there is only one argument in the signature and that the class of the corresponding object was `"dgeMatrix"` (from the recommended package `Matrix`). This class has (currently) three direct superclasses and through these additional superclasses at distances 2 through 4. A method that had been defined for any of these classes or for class `"ANY"` (the default method) would be eligible. Methods for the shortest difference are preferred. If there is only one best method in this sense, method selection is unambiguous.

When there are multiple arguments in the signature, each argument will generate a similar list of inherited classes. The possible matches are now all the combinations of classes from each argument (think of the function `outer` generating an array of all possible combinations). The search now finds all the methods matching any of this combination of classes. For each argument, the distance to the superclass defines which method(s) are preferred for that argument. A method is considered best for selection if it is among the best (i.e., has the least distance) for each argument.

The end result is that zero, one or more methods may be "best". If one, this method is selected and cached in the table of methods. If there is more than one best match, the selection is ambiguous and a message is printed noting which method was selected (the first method lexicographically in the ordering) and what other methods could have been selected. Since the ambiguity is usually nothing the end user could control, this is not a warning. Package authors should examine their package for possible ambiguous inheritance by calling `testInheritedMethods`.

Cached inherited selections are not themselves used in future inheritance searches, since that could result in invalid selections. If you want inheritance computations to be done again (for example, because a newly loaded package has a more direct method than one that has already been used in this session), call `resetGeneric`. Because classes and methods involving them tend to come from the same package, the current implementation does not reset all generics every time a new package is loaded.

Besides being initiated through calls to the generic function, method selection can be done explicitly by calling the function `selectMethod`. Note that some computations may use this function directly, with optional arguments. The prime example is the use of `coerce()` methods by function `as()`. There has been some confusion from comparing coerce methods to a call to `selectMethod` with other options.

Method Evaluation: Details

Once a method has been selected, the evaluator creates a new context in which a call to the method is evaluated. The context is initialized with the arguments from the call to the generic function. These arguments are not rematched. All the arguments in the signature of the generic will have been evaluated (including any that are currently inactive); arguments that are not in the signature will obey the usual lazy evaluation rules of the language. If an argument was missing in the call, its default expression if any will *not* have been evaluated, since method dispatch always uses class `missing` for such arguments.

A call to a generic function therefore has two contexts: one for the function and a second for the method. The argument objects will be copied to the second context, but not any local objects created in a nonstandard generic function. The other important distinction is that the parent ("enclosing") environment of the second context is the environment of the method as a function, so that all R programming techniques using such environments apply to method definitions as ordinary functions.

For further discussion of method selection and dispatch, see the references in the sections indicated.

Generic Functions

In principle, a generic function could be any function that evaluates a call to `standardGeneric()`, the internal function that selects a method and evaluates a call to the selected method. In practice, generic functions are special objects that in addition to being from a subclass of class `"function"` also extend the class `genericFunction`. Such objects have slots to define information needed to deal with their methods. They also have specialized environments, containing the tables used in method selection.

The slots `"generic"` and `"package"` in the object are the character string names of the generic function itself and of the package from which the function is defined. As with classes, generic functions are uniquely defined in R by the combination of the two names. There can be generic functions of the same name associated with different packages (although inevitably keeping such functions cleanly distinguished is not always easy). On the other hand, R will enforce that only one definition of a generic function can be associated with a particular combination of function and package name, in the current session or other active version of R.

Tables of methods for a particular generic function, in this sense, will often be spread over several other packages. The total set of methods for a given generic function may change during a session, as additional packages are loaded. Each table must be consistent in the signature assumed for the generic function.

R distinguishes *standard* and *nonstandard* generic functions, with the former having a function body that does nothing but dispatch a method. For the most part, the distinction is just one of simplicity: knowing that a generic function only dispatches a method call allows some efficiencies and also removes some uncertainties.

In most cases, the generic function is the visible function corresponding to that name, in the corresponding package. There are two exceptions, *implicit* generic functions and the special computations required to deal with R's *primitive* functions. Packages can contain a table of implicit generic versions of functions in the package, if the package wishes to leave a function non-generic but to constrain what the function would be like if it were generic. Such implicit generic functions are

created during the installation of the package, essentially by defining the generic function and possibly methods for it, and then reverting the function to its non-generic form. (See implicitGeneric for how this is done.) The mechanism is mainly used for functions in the older packages in R, which may prefer to ignore S4 methods. Even in this case, the actual mechanism is only needed if something special has to be specified. All functions have a corresponding implicit generic version defined automatically (an implicit, implicit generic function one might say). This function is a standard generic with the same arguments as the non-generic function, with the non-generic version as the default (and only) method, and with the generic signature being all the formal arguments except ....

The implicit generic mechanism is needed only to override some aspect of the default definition. One reason to do so would be to remove some arguments from the signature. Arguments that may need to be interpreted literally, or for which the lazy evaluation mechanism of the language is needed, must *not* be included in the signature of the generic function, since all arguments in the signature will be evaluated in order to select a method. For example, the argument `expr` to the function `with` is treated literally and must therefore be excluded from the signature.

One would also need to define an implicit generic if the existing non-generic function were not suitable as the default method. Perhaps the function only applies to some classes of objects, and the package designer prefers to have no general default method. In the other direction, the package designer might have some ideas about suitable methods for some classes, if the function were generic. With reasonably modern packages, the simple approach in all these cases is just to define the function as a generic. The implicit generic mechanism is mainly attractive for older packages that do not want to require the methods package to be available.

Generic functions will also be defined but not obviously visible for functions implemented as *primitive* functions in the base package. Primitive functions look like ordinary functions when printed but are in fact not function objects but objects of two types interpreted by the R evaluator to call underlying C code directly. Since their entire justification is efficiency, R refuses to hide primitives behind a generic function object. Methods may be defined for most primitives, and corresponding metadata objects will be created to store them. Calls to the primitive still go directly to the C code, which will sometimes check for applicable methods. The definition of "sometimes" is that methods must have been detected for the function in some package loaded in the session and `isS4(x)` is `TRUE` for the first argument (or for the second argument, in the case of binary operators). You can test whether methods have been detected by calling `isGeneric` for the relevant function and you can examine the generic function by calling `getGeneric`, whether or not methods have been detected. For more on generic functions, see the references and also section 2 of the *R Internals* document supplied with R.

Method Definitions

All method definitions are stored as objects from the `MethodDefinition` class. Like the class of generic functions, this class extends ordinary R functions with some additional slots: `"generic"`, containing the name and package of the generic function, and two signature slots, `"defined"` and `"target"`, the first being the signature supplied when the method was defined by a call to `setMethod`. The `"target"` slot starts off equal to the `"defined"` slot. When an inherited method is cached after being selected, as described above, a copy is made with the appropriate `"target"` signature. Output from `showMethods`, for example, includes both signatures.

Method definitions are required to have the same formal arguments as the generic function, since the method dispatch mechanism does not rematch arguments, for reasons of both efficiency and consistency.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (Section 10.5 for some details.)

See Also

For more specific information, see `setGeneric`, `setMethod`, and `setClass`.

For the use of ... in methods, see dotsMethods.

`Methods_for_Nongenerics` *Methods for Non-Generic Functions in Other Packages*

Description

In writing methods for an R package, it's common for these methods to apply to a function (in another package) that is not generic in that package; that is, there are no formal methods for the function in its own package, although it may have S3 methods. The programming in this case involves one extra step, to call `setGeneric()` to declare that the function *is* generic in your package.

Calls to the function in your package will then use all methods defined there or in any other loaded package that creates the same generic function. Similarly, calls to the function in those packages will use your methods.

The original version, however, remains non-generic. Calls in that package or in other packages that use that version will not dispatch your methods except for special circumstances:

1. If the function is one of the primitive functions that accept methods, the internal C implementation will dispatch methods if one of the arguments is an S4 object, as should be the case.
2. If the other version of the function dispatches S3 methods *and* your methods are also registered as S3 methods, the method will usually be dispatched as that S3 method.
3. Otherwise, you will need to ensure that all calls to the function come from a package in which the function is generic, perhaps by copying code to your package.

Details and the underlying reasons are discussed in the following sections.

Generic and Non-Generic Calls

Creating methods for a function (any function) in a package means that calls to the function in that package will select methods according to the actual arguments. However, if the function was originally a non-generic in another package, calls to the function from that package will *not* dispatch methods. In addition, calls from any third package that imports the non-generic version will also not dispatch methods. This section considers the reason and how one might deal with the consequences.

The reason is simply the R namespace mechanism and its role in evaluating function calls. When a name (such as the name of a function) needs to be evaluated in a call to a function from some package, the evaluator looks first in the frame of the call, then in the namespace of the package and then in the imports to that package.

Defining methods for a function in a package ensures that calls to the function in that package will select the methods, because a generic version of the function is created in the namespace. Similarly, calls from another package that has or imports the generic version will select methods. Because the generic versions are identical, all methods will be available in all these packages.

However, calls from any package that imports the old version or just selects it from the search list will usually *not* select methods.

A an example, consider the function `data.frame()` in the base package. This function takes any number of objects as arguments and attempts to combine them as variables into a data frame object. It does this by calling `as.data.frame()`, also in the base package, for each of the objects.

A reasonable goal would be to extend the classes of objects that can be included in a data frame by defining methods for `as.data.frame()`. But calls to `data.frame()`, will still use the version of that function in the base package, which continues to call the non-generic `as.data.frame()` in that package.

The details of what happens and options for dealing with it depend on the form of the function: a primitive function; a function that dispatches S3 methods; or an ordinary R function.

Primitive functions are not actual R function objects. They go directly to internal C code. Some of them, however, have been implemented to recognize methods. These functions dispatch both S4 and S3 methods from the internal C code. There is no explicit generic function, either S3 or S4. The internal code looks for S4 methods if the first argument, or either of the arguments in the case of a binary operator, is an S4 object. If no S4 method is found, a search is made for an S3 method. So defining methods for these functions works as long as the relevant classes have been defined, which should always be the case.

A function dispatches S3 methods by calling `UseMethod()`, which does *not* look for formal methods regardless of whether the first argument is an S4 object or not. This applies to the `as.data.frame()` example above. To have methods called in this situation, your package must also define the method as an S3 method, if possible. See section 'S3 "Generic" Functions'.

In the third possibility, the function is defined with no expectation of methods. For example, the base package has a number of functions that compute numerical decompositions of matrix arguments. Some, such as `chol()` and `qr()` are implemented to dispatch S3 methods; others, such as `svd()` are implemented directly as a specific computation. A generic version of the latter functions can be written and called directly to define formal methods, but no code in another package that does not import this generic version will dispatch such methods.

In this case, you need to have the generic version used in all the indirect calls to the function supplying arguments that should dispatch methods. This may require supplying new functions that dispatch methods and then call the function they replace. For example, if S3 methods did not work for `as.data.frame()`, one could call a function that applied the generic version to all its arguments and then called `data.frame()` as a replacement for that function. If all else fails, it might be necessary to copy over the relevant functions so that they would find the generic versions.

S3 "Generic" Functions

S3 method dispatch looks at the class of the first argument. S3 methods are ordinary functions with the same arguments as the generic function. The "signature" of an S3 method is identified by the name to which the method is assigned, composed of the name of the generic function, followed by `"."`, followed by the name of the class. For details, see `UseMethod`.

To implement a method for one of these functions corresponding to S4 classes, there are two possibilities: either an S4 method or an S3 method with the S4 class name. The S3 method is only possible if the intended signature has the first argument and nothing else. In this case, the recommended approach is to define the S3 method and also supply the identical function as the definition of the S4 method. If the S3 generic function was `f3(x, ...)` and the S4 class for the new method was `"myClass"`:

```
f3.myClass <- function(x, ...) { ..... }

setMethod("f3", "myClass", f3.myClass)
```

Defining both methods usually ensures that all calls to the original function will dispatch the intended method. The S4 method alone would not be called from other packages using the original version of the function. On the other hand, an S3 method alone will not be called if there is *any* eligible non-default S4 method.

S4 and S3 method selection are designed to follow compatible rules of inheritance, as far as possible. S3 classes can be used for any S4 method selection, provided that the S3 classes have been registered by a call to `setOldClass`, with that call specifying the correct S3 inheritance pattern. S4 classes can be used for any S3 method selection; when an S4 object is detected, S3 method selection uses the contents of `extends(class(x))` as the equivalent of the S3 inheritance (the inheritance is cached after the first call).

An existing S3 method may not behave as desired for an S4 subclass, in which case utilities such as `asS3` and `S3Part` may be useful. If the S3 method fails on the S4 object, `asS3(x)` may be passed instead; if the object returned by the S3 method needs to be incorporated in the S4 object, the replacement function for `S3Part` may be useful.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

Methods_for_S3 for suggested implementation of methods that work for both S3 and S4 dispatch.

Examples

```
## A class that extends a registered S3 class inherits that class' S3
## methods.

setClass("myFrame", contains = "data.frame",
         slots = c(timestamps = "POSIXt"))
df1 <- data.frame(x = 1:10, y = rnorm(10), z = sample(letters,10))
mydf1 <- new("myFrame", df1, timestamps = Sys.time())

## "myFrame" objects inherit "data.frame" S3 methods; e.g., for `[`


mydf1[1:2, ] # a data frame object (with extra attributes)


## a method explicitly for "myFrame" class

setMethod("[",
    signature(x = "myFrame"),
    function (x, i, j, ..., drop = TRUE)
    {
        S3Part(x) <- callNextMethod()
        x@timestamps <- c(Sys.time(), as.POSIXct(x@timestamps))
        x
    }
)

mydf1[1:2, ]

setClass("myDateTime", contains = "POSIXt")
```

```
now <- Sys.time() # class(now) is c("POSIXct", "POSIXt")
nowLt <- as.POSIXlt(now)# class(nowLt) is c("POSIXlt", "POSIXt")

mCt <- new("myDateTime", now)
mLt <- new("myDateTime", nowLt)

## S3 methods for an S4 object will be selected using S4 inheritance
## Objects mCt and mLt have different S3Class() values, but this is
## not used.
f3 <- function(x)UseMethod("f3") # an S3 generic to illustrate inheritance

f3.POSIXct <- function(x) "The POSIXct result"
f3.POSIXlt <- function(x) "The POSIXlt result"
f3.POSIXt <- function(x) "The POSIXt result"

stopifnot(identical(f3(mCt), f3.POSIXt(mCt)))
stopifnot(identical(f3(mLt), f3.POSIXt(mLt)))



## An S4 object selects S3 methods according to its S4 "inheritance"

setClass("classA", contains = "numeric",
         slots = c(realData = "numeric"))

Math.classA <- function(x) { (getFunction(.Generic))(x@realData) }
setMethod("Math", "classA", Math.classA)


x <- new("classA", log(1:10), realData = 1:10)

stopifnot(identical(abs(x), 1:10))

setClass("classB", contains = "classA")

y <- new("classB", x)

stopifnot(identical(abs(y), abs(x))) # (version 2.9.0 or earlier fails here)

## an S3 generic: just for demonstration purposes
f3 <- function(x, ...) UseMethod("f3")

f3.default <- function(x, ...) "Default f3"

## S3 method (only) for classA
f3.classA <- function(x, ...) "Class classA for f3"

## S3 and S4 method for numeric
f3.numeric <- function(x, ...) "Class numeric for f3"
setMethod("f3", "numeric", f3.numeric)

## The S3 method for classA and the closest inherited S3 method for classB
## are not found.

f3(x); f3(y) # both choose "numeric" method

## to obtain the natural inheritance, set identical S3 and S4 methods
```

```
setMethod("f3", "classA", f3.classA)

f3(x); f3(y) # now both choose "classA" method

## Need to define an S3 as well as S4 method to use on an S3 object
## or if called from a package without the S4 generic

MathFun <- function(x) { # a smarter "data.frame" method for Math group
  for (i in seq(length = ncol(x))[sapply(x, is.numeric)])
    x[, i] <- (getFunction(.Generic))(x[, i])
  x
}
setMethod("Math", "data.frame", MathFun)

## S4 method works for an S4 class containing data.frame,
## but not for data.frame objects (not S4 objects)

try(logIris <- log(iris)) #gets an error from the old method

## Define an S3 method with the same computation

Math.data.frame <- MathFun

logIris <- log(iris)
```

`Methods_for_S3` *Methods For S3 and S4 Dispatch*

Description

The S3 and S4 software in R are two generations implementing functional object-oriented programming. S3 is the original, simpler for initial programming but less general, less formal and less open to validation. The S4 formal methods and classes provide these features but require more programming.

In modern R, the two versions attempt to work together. This documentation outlines how to write methods for both systems by defining an S4 method for a function that dispatches S3 methods.

The systems can also be combined by using an S3 class with S4 method dispatch or in S4 class definitions. See `setOldClass`.

S3 Method Dispatch

The R evaluator will 'dispatch' a method from a function call either when the body of the function calls the special primitive `UseMethod` or when the call is to one of the builtin primitives such as the `math` functions or the binary operators.

S3 method dispatch looks at the class of the first argument or the class of either argument in a call to one of the primitive binary operators. In pure S3 situations, 'class' in this context means the class

attribute or the implied class for a basic data type such as "numeric". The first S3 method that matches a name in the class is called and the value of that call is the value of the original function call. For details, see S3Methods.

In modern R, a function meth in a package is registered as an S3 method for function fun and class Class by including in the package's NAMESPACE file the directive

```
S3method(fun, Class, meth)
```

By default (and traditionally), the third argument is taken to be the function fun.Class; that is, the name of the generic function, followed by ".", followed by the name of the class.

As with S4 methods, a method that has been registered will be added to a table of methods for this function when the corresponding package is loaded into the session. Older versions of R, copying the mechanism in S, looked for the method in the current search list, but packages should now always register S3 methods rather than requiring the package to be attached.

Methods for S4 Classes

Two possible mechanisms for implementing a method corresponding to an S4 class, there are two possibilities are to register it as an S3 method with the S4 class name or to define and set an S4 method, which will have the side effect of creating an S4 generic version of this function.

For most situations either works, but the recommended approach is to do both: register the S3 method and supply the identical function as the definition of the S4 method. This ensures that the proposed method will be dispatched for any applicable call to the function.

As an example, suppose an S4 class "uncased" is defined, extending "character" and intending to ignore upper- and lower-case. The base function unique dispatches S3 methods. To define the class and a method for this function:

```
setClass("uncased", contains = "character")

unique.uncased <- function(x, incomparables = FALSE, ...)  nextMethod(tolower(x))

setMethod("unique", "uncased", unique.uncased)
```

In addition, the NAMESPACE for the package should contain:

```
S3method(unique, uncased)

exportMethods(unique)
```

The result is to define identical S3 and S4 methods and ensure that all calls to unique will dispatch that method when appropriate.

Details

The reasons for defining both S3 and S4 methods are as follows:

1. An S4 method alone will not be seen if the S3 generic function is called directly. This will be the case, for example, if some function calls unique() from a package that does not make that function an S4 generic.

   However, primitive functions and operators are exceptions: The internal C code will look for S4 methods if and only if the object is an S4 object. S4 method dispatch would be used to dispatch any binary operator calls where either of the operands was an S4 object, for example.

2. An S3 method alone will not be called if there is *any* eligible non-default S4 method.

   So if a package defined an S3 method for unique for an S4 class but another package defined an S4 method for a superclass of that class, the superclass method would be chosen, probably not what was intended.

S4 and S3 method selection are designed to follow compatible rules of inheritance, as far as possible. S3 classes can be used for any S4 method selection, provided that the S3 classes have been registered by a call to `setOldClass`, with that call specifying the correct S3 inheritance pattern. S4 classes can be used for any S3 method selection; when an S4 object is detected, S3 method selection uses the contents of `extends(class(x))` as the equivalent of the S3 inheritance (the inheritance is cached after the first call).

For the details of S4 and S3 dispatch see Methods_Details and S3Methods.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

`MethodWithNext-class` クラス *MethodWithNext*

Description

callNextMethod に対して設定されたメソッド定義のクラス.

クラスからのオブジェクト

このクラスからのオブジェクトが `callNextMethod` の呼び出しの副作用として生成される.

スロット

`.Data`: クラス `"function"` のオブジェクト；実際の関数定義.

`nextMethod`: クラス `"PossibleMethod"` のオブジェクト；呼び出し `callNextMethod()` に対応して使われるメソッド.

`excluded`: クラス `"list"` のオブジェクト；次のメソッドを見つける際に除外される一つまたはそれ以上のシグネチャ.

`target`: クラス `"signature"` のオブジェクト，クラス `"MethodDefinition"` から

`defined`: クラス `"signature"` のオブジェクト，クラス `"MethodDefinition"` から

`generic`: クラス `"character"` のオブジェクト；それに対してメソッドが作られた関数.

拡張

クラス `"MethodDefinition"`，直接に.
クラス `"function"`，データ部分から.
クラス `"PossibleMethod"`，クラス `"MethodDefinition"` により.
クラス `"OptionalMethods"`，クラス `"MethodDefinition"` により.

メソッド

findNextMethod `signature(method = "MethodWithNext")`: メソッドの選択適用により内部的に使われる.

loadMethod `signature(method = "MethodWithNext")`: メソッドの選択適用により内部的に使われる.

show `signature(object = "MethodWithNext")`

See Also

callNextMethod とクラス MethodDefinition.

---

new　　クラスからオブジェクトを生成する

Description

クラスの名前か定義とオブジェクト中に含められるオプションのデータを与え，new はそのクラスからのオブジェクトを返す．

Usage

```
new(Class, ...)

initialize(.Object, ...)
```

Arguments

Class	クラスの名前か character 文字列 (通常のケース)，またはクラスを記述するオブジェクト (つまり getClass が返すもの)．
...	新しいオブジェクト中に含められるデータ．クラス定義中のスロットに対応する名前付き引数．名前無しの引数はこのクラスが拡張するクラスからのオブジェクトでなければならない．
.Object	オブジェクト：詳細節を見よ．

Details

関数 new はクラス定義からプロトタイプオブジェクトをコピーすることから始める．それから，もしあれば情報が ... 引数に従い挿入される． R のバージョン 2.4 現在，プロトタイプオブジェクト，そして従って new() が返す全てのオブジェクトは，プロトタイプがその基本タイプを持つ時基本タイプの一つを拡張するクラスを例外として "S4" である． typeof(object) に依存するユーザ関数は可能なタイプとして "S4" を処理することに注意すべである．

最初の引数の名前 "Class" は "Class" が任意の形式的クラス中の好ましくないスロット名であることを必須とすることを注意する： new("myClass", Class = <value>) は使える．

... 引数の解釈は，もし総称的関数 "initialize" に対する適当なメソッドが定義されていれば，特定のクラスに特殊化出来る． new 関数は initialize を initialize への .Object 引数としてプロトタイプから生成されたオブジェクトと共に呼び出される．

既定では， ... 中の名前のない引数はスーパークラスからのオブジェクトと解釈され，名前付き引数は対応する名前付きスロット中に付値されるオブジェクトと解釈される．このように，明示的なスロットは引数が現れる順序と無関係に，同じスロットに対する継承情報を上書きする．

initialize メソッドは ... をそれらの第二引数として持つべきではない (例を見よ)．新しいオブジェクトを記述する自然なパラメータがスロット名では無い時初期化メソッドがしばしば書かれる．もしそうしたメソッドを定義するならば，自分のクラスの将来のサブクラスに対する含意を注意する．もしこれらが追加のスロットを持ち，自分の initialize メソッドが ... を形式的引数として持つならば，メソッドはそうした引数を

callNextMethod を使って渡すべきである．もし自分のメソッドがこの引数を持たなければ，サブクラスがそれ自体のメソッドを持つか，または付け加えられたスロットが new への引数として以外の方法でユーザにより指定されなければならない．

initialize メソッドに対する例としては，とりわけ既存メソッドに対する initialize-methods クラス "traceable" と "environment" を見よ．そこの "environment" のサブクラスに関するコメントを見よ；これらに対する任意の initialize メソッドは新しい環境を確保することが確実であるべきである．

メソッドは目標のクラスからのオブジェクトを返すことが要求されるので， initialize に対するメソッドは単純な継承によってのみ継承される． setGeneric に対する simpleInheritanceOnly 引数と，一般的な概念に対する setIs 中の議論を見よ．

"numeric" 等の基本ベクトルクラスは暗黙のうちに定義されるのでこれらのクラスに対して new を使うことが出来ることを注意する．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

クラス定義の概観は Classes，そして S3 クラスとの関連に対しては setOldClass.

Examples

```
## \link{setClass} からのクラス "track" を使う



## 次で指定される二つのスロットを持つ新しいオブジェクト
t1 <- new("track", x = seq_along(ydata), y = ydata)

# スーパークラスからのオブジェクトと，次のスロットを含む新しいオブジェクト
t2 <- new("trackCurve", t1, smooth = ysmooth)

### 初期化のためのメソッドを定義,
### 新しいオブジェクトが等しい長さの x と y スロットを持つことを保証する.

setMethod("initialize",
          "track",
          function(.Object, x = numeric(0), y = numeric(0)) {
            if(nargs() > 1) {
              if(length(x) != length(y))
                stop("specified x and y of different lengths")
              .Object@x <- x
              .Object@y <- y
            }
            .Object
          })

### x に対する既定値を組み込んでいないため，次の例はエラーになる
### (x は numeric(0) になる),
### 初期化のためのより洗練されたメソッドを使うことができたであろうが
```

```
try(new("track", y = sort(stats::rnorm(10))))

## 先の初期化メソッドを実装するより良い方法.
## 何故か? 既定の初期化メソッドを呼び出すため callNextMethod を使い,
## "track" を拡張するクラスが new() 関数の一般的な書式を使うことを禁止しない.
## 先のバージョンでは, それら自体の初期化メソッドを欠いていない限り,
## それらは x と y を new への引数としてだけ使える

setMethod("initialize", "track", function(.Object, ...) {
    .Object <- callNextMethod()
    if(length(.Object@x) != length(.Object@y))
     stop("specified x and y of different lengths")
    .Object
  })
```

---

nonStructure-class　　基本タイプに対する非構造的 *S4* クラス

Description

基本ベクトルクラスの一つを拡張するために定義された S4 クラスは, もしそれらが構造として動作するならば, クラス structure を含まなければならない ; つまりもしそれらが数学関数や演算の下でそれらのクラス挙動をそれらの長さが未変更である限り保つべきであれば. 一方で, もしそれらのクラスが単にその構造だけでなくオブジェクト中の値に依存するならば, それらは任意のそうした変換の下でそのクラスを失うべきである. 後者の場合, それらは nonStructure を含むように定義されるべきである.

もしこれらの戦略のどれもが適用されなければ, クラスは Ops, Math そして/または他の総称的関数に対するそれ自体のある種のメソッドを必要とする可能性が高い. 普通良いアイデアで無いことは既定の基礎コードに落とすような計算を認めることである. これはそうしたクラスの殆どの定義と一貫性が無い.

メソッド

演算子と数学関数 (グループ Ops, Math そして Math2) に対するメソッドが定義されている. 全てのケースで結果は適当なタイプの通常のベクトルである.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer.

See Also

structure

Examples

```
setClass("NumericNotStructure", contains = c("numeric","nonStructure"))
xx <- new("NumericNotStructure", 1:10)
xx + 1 # ベクトル
log(xx) # ベクトル
sample(xx) # ベクトル
```

ObjectsWithPackage-class
関連するパッケージ名を持つオブジェクト名のベクトル

Description

このオブジェクトのクラスは通常の文字列オブジェクト名を表現するのに使われ，各オブジェクトに関連するパッケージの名前を与える package スロットで拡張されている．

クラスからのオブジェクト

関数 getGenerics はこのクラスのオブジェクトを返す．

スロット

.Data: クラス "character" のオブジェクト：オブジェクト名．

package: クラス "character" のオブジェクト，パッケージ名．

拡張

クラス "character"，データ部分から．
クラス "vector"，クラス "character" により．

See Also

一般的な背景については Methods．

---

promptClass 形式的クラスのドキュメントに対するシェルを生成する

Description

全ての関連するスロットとメソッド情報を収集し， Rd 処理のための最少のマークアップを持つ；現在 QC のための機能は無い．

Usage

```
promptClass(clName, filename = NULL, type = "class",
            keywords = "classes", where = topenv(parent.frame()),
            generatorName = clName)
```

Arguments

clName	ドキュメントされるクラスの名前を与える文字列．
filename	普通ドキュメントシェルが書き込まれるべきコネクションまたはファイル名．既定では名前がクラスドキュメントに対するトピック名であるファイルで， ".Rd" が続く．NA でも良い(下を見よ)．
type	出力ファイル中で宣言されるドキュメントタイプ．

keywords ドキュメントのシェル中に含まれるキーワード．キーワード "classes" はそれらの一つでなければならない．

where クラスとそれを使うメソッドの定義をどこで探すか．

generatorName このクラスに対する生成関数に対する名前；もし生成関数が作られそしてクラス名と異なる名前で保存された時だけ必要．

Details

クラス定義は検索パス上で見つけられる．その定義を使い拡張されたクラスとスロットの情報が決定される．

加えて，このクラスに対するメソッドを持つ現在利用可能な総称的関数が見つけられる(getGenerics を使う)．これらのメソッドはクラス定義と同じ環境にある必要はない；特に，出力のこの部分はどのパッケージが現在検索パス上にあるかに依存する．

他のプロンプトスタイルの関数のように， filename が NA で無い限り，ドキュメントシェルはファイルに書き込まれそしてこれに対するメッセージが与えられる．クラスの意味に関する情報を与えるにはファイルを編集する必要がある． promptClass の出力は形式的定義とそれがどのように使われるかに関するメタデータからの情報だけを含むことが出来る．

もし filename が NA ならば，ドキュメントシェルのリストスタイルの表現が作られ返される．シェルをファイルに書き込むことは cat(unlist(x), file = filename, sep = "\n") に相当する，ここで x はリストスタイルの表現である．

もし生成関数がクラス名かオプションの generatorName に付値されて見つかれば，その関数に対するドキュメント骨格がファイルに付け加えられる．

Value

もし filename が NA ならば，ドキュメントシェルのリストスタイルの表現．さもなければ，書き込まれたファイルの名前が不可視で返される．

Author(s)

VJ Carey <stvjc@channing.harvard.edu> と John Chambers

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

関数のドキュメントに対しては prompt，メソッドの定義のドキュメントに対しては promptMethods.

編集されたドキュメントの処理に対しては R CMD Rdconv を使うか，パッケージの 'man' サブディレクトリ中に編集されたファイルを置く．

Examples

```
## Not run: > promptClass("track")
ファイル "track-class.Rd" にクラスドキュメントのシェルが書き込まれている.

## End(Not run)
```

---

promptMethods 形式的メソッドのドキュメントに対するシェルを生成

Description

総称的関数のメソッドのドキュメントに対するシェルを生成する.

Usage

```
promptMethods(f, filename = NULL, methods)
```

Arguments

f	そのメソッドがドキュメント化されるべき総称的関数の名前を与える文字列.
filename	普通コネクションかドキュメントのシェルがそこに書き込まれるファイルの名前を与える文字列. 既定はこれらのメソッドのコード化されたトピック名に対応する (現在 f に "-methods.Rd" が続く). FALSE や NA でも良い(下を見よ).
methods	ドキュメント化されるべきメソッドを与えるオプションのメソッドのリスト. 既定では, この総称的関数に対する最初のメソッドが使われる (例えば, もし現在の大局的環境が f に対するあるメソッドを持てば, それらがドキュメント化される). もしこの引数が提供されると, それは恐らく getMethods(f, where) で, where は f に対するメソッドを含むあるパッケージである.

Details

もし filename が FALSE ならば, 作られたテキストが返され, もしかすると総称的関数自体のドキュメントのような (prompt を見よ)他のドキュメントファイルが挿入されるかもしれない.

もし filename が NA ならば, ドキュメントのシェルのリスト風の表現が作られ返される. シェルをファイルに書き込むことは cat(unlist(x), file = filename, sep = "\n") に相当する, ここで x はリスト風表現である.

さもなければ, ドキュメントのシェルは filename で指定されるファイルに書き込まれる.

Value

もし filename が FALSE ならテキストが生成される ; もし filename が NA ならば, ドキュメントのシェルのリスト風の表現が生成される. さもなければ書き込まれるファイルの名前が不可視で返される.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

`prompt` と `promptClass`

`ReferenceClasses` 参照として扱われる欄を持つオブジェクト*(OOP* スタイル*)*

Description

ここで説明されるソフトウェアは Java と C++ のような “OOP” 言語スタイルでの参照によりアクセスされる欄を持つオブジェクトの参照クラスをサポートする．

これらのオブジェクトを使った計算はそれらに対するメソッドを起動しそれらの欄を取り出したり設定したりする．欄とメソッド計算は可能性としてオブジェクトを修正する．オブジェクトを参照する全ての計算は通常の R 中の関数プログラミングモデルとは異なり修正を見る．参照クラスは R 中や OOP スタイルの言語へのインタフェイスとの連携で使うことが出来，インタフェイスを拡張する R で書かれたメソッドを許す．

Usage

```
setRefClass(Class, fields = , contains = , methods =,
     where =, inheritPackage =, ...)

getRefClass(Class, where =)
```

Arguments

`Class`	クラスに対する文字列名． `getRefClass()` の呼び出し中ではこの引数は関連クラスからの任意のオブジェクトでも良い；“参照メソッドを書く”節中に説明された対応する参照クラスメソッドも注意せよ．
`fields`	欄名の文字列ベクトルか欄の名前付きリスト．結果の欄は参照のセマンティックスを使いアクセスされる (“参照オブジェクト”についての節を見よ)．もし引数がリストなら，リストの要素はクラスの文字列名で良く，その場合欄はそのクラスかサブクラスからのものでなければならない． リスト中の要素はまたは アクセス関数でもよく，この関数は引数一つの関数で引数無しで呼ばれると欄を返し，さもなければ引数の値で欄を設定する．アクセス関数は内部的とシステム間のインタフェイスアプリケーションに対して使われる．これらの定義はクラスに対するメソッドを書くための規則に従う；これらは他の欄を参照でき，このクラスやそのスーパークラスに対する他のメソッドを呼び出すことが出来る．アクセス関数により使われる内部的機構に対しては節“実装”を見よ．

欄はもしあればオブジェクト中のスロットとは異なることを注意する．スロットは通常のように標準的な R のオブジェクト管理で処理される．同じクラス中にスロットと欄を混在させるのは一般に良いアイデアでは無い；これは参照クラスと正規の S4 クラス間の挙動間の違いを混乱させる． “実装”節中のコメントを見よ．

`contains` このクラスに対するスーパークラスのオプションのベクトル．もしスーパークラスがまた参照クラスならば，欄とクラスベースのメソッドが継承される．

`methods` このクラスからのオブジェクトに適用できる関数定義の名前付きリスト．これらは又返される生成器オブジェクトに `$methods` メソッドを呼び出すことで作ることが出来る．詳細は節“参照メソッドを書く”を見よ．

二つのオプションのメソッド名もし `initialize` メソッドが定義されると，それはそのクラスからオブジェクトが生成されると適用される．節“参照オブジェクト生成器”中のメソッド `$new(...)` の議論を見よ．

もし `finalize` メソッドが定義されると，関数はオブジェクト中の環境がガベージコレクションで捨て去られる前に呼び出されるように registered される；終結化は `atexit=TRUE` で登録されるので， R セッションの最後にもまた実行される．初期化と終結化メソッドの双方については行列ビューアの例を見よ．

`where` クラス定義をその中に保存 (または `getRefClass()` の場合はそれの探索を始める)する環境．既定ではパッケージの名前空間か R パッケージの部分であるコードに対する環境と，セッションのトップレベルで直接読み込まれたコードに対する大局的環境．

`inheritPackage` 新しいクラスからのオブジェクトは含まれるスーパークラスのパッケージ環境を継承すべきか? 既定値は `FALSE`． 節 “パッケージ間のスーパークラスと外部メソッド”.

`...` `setClass` に渡される他の引数．

Value

`setRefClass()` はクラスからのオブジェクトの作成に適した生成関数を不可視で返す．この関数の呼び出しは任意の数の引数を取る．クラスやそのスーパークラスの一つにに対する `initialize` メソッドが適用されていなければ，既定のメソッドは欄の一つの名前を持つ名前付き引数と，もしあれば，このクラスのスーパークラスの一つからのオブジェクトである名前無しの引数を取る (しかしそれ自体が参照クラスであるスーパークラスだけが何らかの効果を持つ).

生成関数は `setClass` が返す S4 生成関数に類似するが，同時に参照クラス生成オブジェクトであり，様々なユーティリティに対する参照クラスメソッドを持つ．下の参照クラス生成オブジェクトに関する節を見よ．

もしクラスが `$initialize()` に対して定義されたメソッドを持てば，このメソッドは参照オブジェクトが作られた時一度呼び出される．ある種の特殊な初期化を行う必要があるクラスに対してそうしたメソッドを書くべきである．特に，参照クラスに対しては欄の初期化の前に参照オブジェクトに対するある種の特別な初期化が必要になるため， S4 総称的関数 `initialize()` に対するメソッドではなく参照メソッドが推奨される．S4 クラスに於けるように，先の理由と又 `$new()` は生成オブジェクトに対して適用されそのクラスに対するメソッドになるための双方から，メソッドは `$new()` ではなく `$initialize()` に対して書かれる．

`$initialize()` に対する既定メソッドはメソッド `$initFields(...)` の適用と同値である．名前付き引数は対応する欄の初期値を付値する．名前無し引数はこのクラスかその

スーパークラスからのオブジェクトでなければならない．欄はそうしたオブジェクトの欄の内容に初期化されるが，名前付き引数は対応する継承欄を上書きする．欄は単純に付値されることを注意する．もし欄がそれ自体参照オブジェクトならば，そのオブジェクトはコピーされない．新しいオブジェクトと以前のオブジェクトは参照を共有する．又，名前無し引数から付値された欄はロックされた欄に対する付値であるとカウントされる．ロックされた欄に対する継承値を上書きするには，新しい値は初期化呼び出し中の名前付き引数の一つでなければならない．欄の後からの付値はエラーになる．

初期化メソッドのデザインはある配慮を必要とする．参照クラスに対する生成器は例えばオブジェクトをコピーする際引数無しで呼び出される．これらの呼び出しが失敗しないことを保証するには，メソッドは全ての引数に対する既定値を持つか `missing()` に対するチェックを必要とする．メソッドは引数として `...` を含むべきであり，これを `$callSuper()` (またはもし自分のスーパークラスが初期化メソッドを持たないことがわかっていれば `$initFields()`) を使って渡すべきである．これはこのクラスをサブクラスとする追加の欄を持つ将来のクラス定義を可能にする．

`getRefClass()` は又クラスに対する生成関数を返す．必要ならクラスを見つけるためだけに使われる `where` 引数とは無関係に，値中のパッケージスロットはクラス定義からの正しいパッケージであることを注意する．

参照オブジェクト

R 中の普通のオブジェクトは関数呼び出しに於いて関数プログラミングのセマンティックスと一貫性を持って引数として渡される；つまり，引数として渡されるオブジェクトに加えられた変更は関数呼び出しに対して局所的である．引数を提供したオブジェクトは変更されない．

関数モデル(R ではこれは不正確ではあるがしばしば値渡しと呼ばれる)は例えば統計モデル当てはめに対する基本 R ソフトウェアのような，多くの統計計算に適している．ある種の他の状況では，任意の計算中で行われた変更がどこでも反映されるように，あるオブジェクトを扱う全てのコードが正確に同じ内容を見ることが好ましい．これはユーザインタフェイス中のウィンドウのようにオブジェクトがある“客観的”な実在性を持つならばしばしば好ましい．

加えて， Java, C++ そして多くの他を含むよく使われる言語は参照セマンティックスを仮定するクラスとメソッドのバージョンを持つ．対応するプログラミング機構はオブジェクトに対してメソッドを適用する． R のシンタックス中では，この操作に対して `"$"` を使う；例えばオブジェクト `x` に対してメソッド `m1` を表現式 `x$m1(...)` で適用する．

このパラダイムにおけるメソッドはメソッド，より正確に言えばオブジェクトのクラスに関連しており，基本的に関数に関連する関数ベースのクラス/メソッドシステム中のメソッドと対比される (R では例えばある R セッション中の総称的関数はその現在知られているメソッドのテーブルを持つ)．このドキュメントでは “関数に対するメソッド”に対する “クラスに対するメソッド”と区別される．

このパラダイムにおけるオブジェクトは普通それに対してメソッドが操作される名前付き欄を持つ． R の実装では，欄はクラスが作成される時定義される．欄自体はオプションで指定されたクラスを持つことが出来，このクラスかそのサブクラスの一つからのオブジェクトだけが欄に付値出来ることを意味する．既定では，欄はクラス `"ANY"` を持つ．欄は又欄を取り出したり設定したりするために呼び出されるアクセス関数を提供することで定義できる．アクセス関数は参照クラスがシステム間インタフェイスの一部である時に適当である．インタフェイスは普通他の言語中の対応するクラスの定義に基づくアクセス関数を自動的に提供する．

欄は参照によりアクセスされる．特に，メソッドの適用は欄の内容を変更するかもしれない．

そうしたクラスに対するプログラミングは特定のクラスに対する新しいメソッドを書くことを含む． R の実装に於いては，これらのメソッドは R 関数で，ゼロ個またはそれ以

上の引数を持つ．メソッドがそれに対して適用されるオブジェクトはメソッドに対する明示的な引数ではない．代わりに，クラスに対する欄とメソッドはメソッド定義中の名前により参照することが出来る．実装は名前で利用できる欄とメソッドを作るために R 環境を利用する．追加の特殊な欄は完全なオブジェクトとクラスの定義への参照を許す．節“参照メソッドを書く”を見よ．

ここで説明されたソフトウェアのゴールは， R 中で直接実装されるにせよ OOP 言語のどれかへのインタフェイスであるにせよ，参照クラスを扱うソフトウェアに対する R の一様なプログラミングスタイルを提供することである．

参照メソッドを書く

参照メソッドは `g$methods()` を生成オブジェクト `g` に対して適用する時かまたは `setRefClass` の呼び出し中に引数 `methods` として名前付きリストとして提供される関数である．それらは通常の R 関数として書かれるがある種の特殊な特徴と制限を持つ．関数の本体は任意の他の参照メソッドへの呼び出しを含むことが出来，他の参照クラスから継承されたものを含みオブジェクト中の欄を名前で参照するかもしれない．

または，メソッドはその第一引数が `.self` である外部メソッドかもしれない．

そうしたメソッドの本体は通常の関数として動作する．メソッドは他のメソッドと同様に呼び出される (内部的に提供され常にオブジェクト自体を参照する `.self` 引数無しに)．外部メソッドは参照クラスが他のパッケージ中のスーパークラスのパッケージ環境を継承できるように存在する；節“パッケージ間スーパークラスと外部メソッド”を見よ．

欄はメソッド中で下の例中の `$edit` と `$undo` に於けるように非局所的な付値演算子 `<<-` で修正できる．非局所的な付値が必要とされることを注意する： `<-` 演算子を用いた局所的な付値は任意の R 関数中でそうあるべきであるように関数呼び出し中で局所的なオブジェクトを作る．メソッドがインストールされる時，欄名に対する局所的な付値に対する発見法的なチェックが行われ，もしあれば警告が発せられる．

参照メソッドは単純であるべきである；もしそれらがある特殊な R 計算を必要とするならば，計算は参照メソッド中から呼び出される別個の R 関数を使うべきである．具体的に言うと，メソッドの環境は欄と他のメソッドにアクセスするために使われるため，メソッドは囲み環境機構の特殊な特性を使うことが出来ない．特に，メソッドはパッケージの名前空間中の移出されない項目を使うべきではない，なぜならメソッドは別のパッケージ中の参照クラスにより継承されているかもしれないからである．

`$initialize()` に対するメソッドは特殊な要求を持つ． “値”切中のコメントを見よ．

参照メソッドはそれら自体総称的関数にはなれない；もし追加の関数ベースのメソッド選択適用が必要ならば，別個の総称的関数を書きそれをメソッドから呼び出す．

オブジェクト全体は例の `save=` メソッド中で示されたようにメソッド中で予約語 `.self` で参照できる特殊オブジェクト `.refClassDef` はオブジェクトのクラスの定義を含む．これらの欄は一つの例外を除いて読み取り専用である (これらの参照を修正することは意味が無い)．原則として， `.self` 欄は `$initialize` メソッド中で変更できる．この段階ではオブジェクトは依然として構築中だからである．ある種のこのクラスに対して定義されたオブジェクトの非参照特性を設定するのでなければこれは明確に推奨できない．これ自体もしそれにスロットと欄が混在するならば推奨できない．

次の節で議論されるように利用できるメソッドはスーパークラスから継承されるメソッドを含む．

実際に使われるメソッドだけが個別のオブジェクトに対応する環境中に含められる．あるメソッドが特定の他のメソッドを必要とすることを宣言するには，最初のメソッドは `$usingMethods()` に呼び出しを他のメソッドの名前を引数として含むべきである．メソッドをこのように宣言することはもし他のメソッドが間接的に使われるときは本質的である (例えば `sapply()` か `do.call()` 経由で)．もしそれが直接呼び出されると，コード解

析がそれを見つける．メソッドの宣言はいかなる場合も無害であるが，ソースコードの可読性を助けるかもしれない．

メソッドのドキュメント化は生成オブジェクトに対する `$help` メソッドで得ることが出来る．クラスに対するメソッドは R 関数に対して使われる `Rd` 書式ではドキュメント化されない．代わりに，`$help` メソッドはメソッドの呼び出し系列をプリントし，メソッド定義からの Python 風の自己ドキュメントが後に続く．もしメソッドの本体の最初の要素が文字通りの文字列 (可能性として複数行)ならば，文字列はドキュメントとして解釈される．例中のメソッド定義を見よ．

継承

参照クラスは標準の R の継承を使うことで他の参照クラスを継承する；つまり，新しいクラスを作る際に `contains=` 引数にスーパークラスを含む．参照スーパークラスの名前はクラス定義の `refSuperClasses` スロット中にある．参照クラスは通常の S4 継承できるが，しかしこれはもしそれが参照欄と非参照スロットを併せ持てば普通悪いアイデアである．“実装”に関する節中のコメントを見よ．

クラスの欄は継承される．クラス定義はスーパークラス中の同名の欄を書き換えクラスが継承欄のクラスのサブクラスである時だけ上書きする．これは欄中の適正なオブジェクトがスーパークラスに対しても同様に適正であることを保証する．

継承メソッドは直接に指定されたメソッドと同じ仕方でインストールされる．メソッド中のコードは直接に指定されたメソッドと同じ仕方で継承メソッドを参照できる．

メソッドはスーパークラス中の同名のメソッドを上書きするかもしれない．上書きするメソッドはスーパークラスメソッドを下で解説されるように `callSuper(...)` によって呼び出すことが出来る．

全ての参照クラスは以下のメソッドを提供するクラス `"envRefClass"` を継承する．

`$callSuper(...)` 参照スーパークラスから継承されたメソッドを呼び出す．呼び出しは別のメソッド内からだけ意味を持ち，同名の継承メソッドの呼び出しになる．`$callSuper` への引数はスーパークラスバージョンに渡される．例中の行列ビューアクラスを見よ．

スーパークラスメソッドに対する意図する引数は明示的に提供される必要があることを注意する；関数メソッドに対する同様の機構とは対照的に，引数を自動的に提供するための変換は無い．

`$copy(shallow = FALSE)` オブジェクトのコピーを作る．参照クラスに対しては，通常の R オブジェクトとは異なり，オブジェクトを異なった名前で付値することは独立なコピーを作らない．もし `shallow` が `FALSE` ならば，それ自体参照オブジェクトである任意の欄もコピーされ，その欄に対しても同様に再帰的にコピーされる．さもなければ，新しい参照オブジェクトへの欄の再付値は副作用を持たず，そうした欄の修正はオブジェクトのコピーのどちらにも依然として反映される．引数は欄中の非参照的なオブジェクトには何も効果を持たない．ある欄に参照オブジェクトがあるがそれらが修正されるべきではないと保証されているときは， `shallow = TRUE` の使用が幾らかメモリーと時間を節約する．

`$field(name, value)` 引数が一つならば文字列 `name` を持つオブジェクトの欄を返す．引数が二つならば，対応する欄に `value` を付値する．付値は `name` が適正な欄を指定することをチェックするが，引数が一つのバージョンはオブジェクトの環境からその名前を持つ何かを得ようと試みる．

`$field()` メソッドは，欄の名前が計算されなければならない時や幾つかの欄に渡るループに対して，欄名の直接的な使用を置き換える．

`$export(Class)` オブジェクトの `Class` への強制変換 (典型的にはオブジェクトのスーパークラスの一つ)の結果を返す．メソッドの呼び出しはオブジェクト自体には何らの副作用を持たない．

`$getRefClass()`; `$getClass()` これらはそれぞれこのオブジェクトの参照クラスに対する生成オブジェクトと形式的な定義を効率的に返す.

`$import(value, Class = class(value))` 現在のオブジェクト中の対応する欄を置き換え，オブジェクト `value` を現在のオブジェクトに移入する．オブジェクト `value` は現在のオブジェクトのクラスのスーパークラスの一つに由来しなければならない．もし引数 `Class` が提供されると， `value` は先ずそのクラスに強制変換される．

`$initFields(...)` オブジェクトの欄を提供された引数から初期化する．このメソッドは普通 `$initialize()` メソッドを持つクラスだけから呼び出される．参照クラスに対してはそれは既定の初期化に対応する．もしスロットと非参照的スーパークラスがあれば，これらも `...` 引数から提供できる．

典型的には，特殊化された `$initialize()` メソッドはそれ自体の計算を行い，それから標準的な初期化を実行するために下の例の `matrixViewer` クラス中で示されるように `$initFields()` を適用する．

`$show()` このメソッドは `show` 関数と同じようにオブジェクトが自動的にプリントされる時に呼び出される．クラス "envRefClass" に対して一般的なメソッドが定義されている．ユーザ定義の参照クラスはしばしばそれら自体のメソッドを定義する：下の例を見よ．

例の中の二つの点に注意する．任意の `show()` メソッドに於けるように，サブクラスに対してメソッドを用いることを認めるためクラスを明示的にプリントするのは良いアイデアである．次に，メソッドから 関数 `show()` を呼び出すには， `$show()` メソッド自体とは異なり， `methods::show()` を明示的に参照する．

`$trace(what, ...)`, `$untrace(what)` `trace` 関数のトレースとデバッグ機能を参照メソッド `what` に適用する．

意味のない `signature` 引数を除き， `trace` 関数への全ての引数が提供できる．

参照メソッドはオブジェクトとしてもクラスに対する生成器としても適用できる．詳細は下のデバッグの節を見よ．

`$usingMethods(...)` このメソッドにより使われる参照メソッドは引用符付きでも無しでも引数として名前を持つ．現在のメソッドのインストールのコード分析フェーズ中では，宣言されたメソッドが含められる．非標準的な仕方で使われる任意のメソッドを宣言することが不可欠である(例えば apply 関数を使い)．直接呼ばれるメソッドは宣言される必要はないが，そうすることは無害である． `$usingMethods()` は実行時には何もしない．

オブジェクトはまた二つの予約欄を継承する：

`.self` オブジェクト全体への参照；

`.refClassDef` クラス定義.

定義された欄はこれらを上書きすべきではなく，そして一般的に実装が特殊な目的のためにそうした名前を使うかもしれないので，名前が "." で始まる欄を定義することは賢明ではない.

パッケージ間スーパークラスと外部メソッド

参照クラス中のメソッドの環境は環境としてのオブジェクト自体である．これはオブジェクト全体と "$" 演算子無しに，欄と他のメソッドを直接に参照することを許す．その環境の親はその中に参照クラスが定義されているパッケージの名前空間である．メソッド中の計算は，移出されているかどうかを問わず，パッケージの名前空間中の全てのオブジェクトへのアクセスを持つ.

別のパッケージ中の参照スーパークラスを含むクラスを定義する時，どのパッケージの名前空間がその役目を持つべきかについて曖昧さがある． `setRefClass()` の引数

inheritPackage は新しいオブジェクトの環境が別のパッケージ中の継承クラスを継承すべきか，現在のパッケージの名前空間から継承することを続けるべきかを制御する．

もしスーパークラスが僅かなメソッドを持ち“傾いて”いるか，主としてスーパークラスのファミリをサポートするために存在していいれば，新しいパッケージの環境を使い続けるほうがよいかもしれない．他方で，もしスーパークラスが本来単独使用のために書かれていれば，この選択は既存のスーパークラスメソッドを損なうかもしれない．スーパークラスメソッドが動作し続けるためには，それらはそれらのパッケージ中の移出関数だけを使わなければならず，そして新しいパッケージはこれらを移入しなければならない．

どちらの方法でも，ある種のメソッドは参照クラスメソッドに対する標準モデルを仮定しない がしかし本質的に通常の関数が参照クラスオブジェクトを処理するように挙動するように書かれる必要があるかもしれない．

機構は 外部メソッドを認識する．外部メソッドは名前付きの .self が参照クラスオブジェクトを表す第一引数の関数として書かれる．この関数は参照クラスメソッドに対する定義として提供される．メソッドは自動的に呼びだされ，最初の引数は現在のオブジェクトで，もしあれば他の引数は実際の呼び出しに合わせて渡される．

外部メソッドはそのパッケージに対するソースコード中の通常の関数であるので，名前空間中の全てのオブジェクトにアクセスする．参照クラス中の欄とメソッドは .self$name という形式で参照されなければならない．

もしある理由で .self を第一引数として使いたくなければ，関数 f() は明示的に externalRefMethod(f) として変換可能で，これはメソッドに対して提供することが出来るクラス "externalRefMethod" のオブジェクトを返す．最初の引数は依然としてオブジェクト全体に対応する．

外部メソッドは任意の参照クラスに対して提供できるがそれらが必要でない限り明白な利点はない．それらは書くのが難しく，可読性が低く，そして (少々)実行が遅い．

注意：もしその参照クラスが恐らく別のパッケージ中のサブクラスのパッケージの作者ならば，自分のパッケージからの移出関数だけを使うメソッドを書くことでこれらの問題を完全に避けることが出来るので，全てのメソッドが自分のそれらを移入する別のパッケージから動作する．

参照クラスの生成器

setRefClass の呼び出しは特殊なクラスを定義し，そしてそのクラスに対する“生成関数”オブジェクトを返す．このオブジェクトはクラス "refObjectGenerator" を持つ；それは "function" を "classGeneratorFunction" を通じて継承し，そして参照クラスからの新しいオブジェクトを生成するために呼び出すことが出来る．

返されたオブジェクトはまた参照クラスオブジェクトであるが，標準的な構成ではない．それは通常の仕方で参照メソッドを適用し欄にアクセスするために使うことができるが，環境として直接実装される代わりに標準的な参照オブジェクト(クラス "refGeneratorSlot") のスロットとしての補助的な生成オブジェクトを持つ．もしサブクラスを持つ参照クラス生成機能を拡張したければ，これは "refObjectGenerator" ではなく "refGeneratorSlot" をサブクラスとして行われるべきである．

欄はクラス定義である def とクラスの文字列名である className である．メソッドは，参照メソッドに関するヘルプにアクセスし，そしてクラスに対する新しい参照メソッドを定義するクラスからのオブジェクトを生成する．現在利用できるメソッドは：

$new(...) このメソッドは setRefClass により返される生成関数の呼び出しと同値である．

$help(topic) トピックに関するヘルプを簡潔にプリントする．認識されるトピックスは参照メソッド名で，引用化されていてもされていなくても良い．

プリントされる情報はメソッドの呼び出し系列で，もしあれば自己ドキュメントがプラスされる．参照メソッドはメソッドを定義する関数の本体中の第一引数として初期文字列またはベクトルを持つことが出来る．もしそうなら，この文字列はメソッドに対する自己ドキュメントとされる (詳細は節“参照メソッドを書く”を見よ).

トピックが与えられないかトピックがメソッド名でないと，クラスのメソッドがプリントされる．

`$methods(...)` 引数がないと，このクラスに対する参照メソッドの名前を返す．一つの文字列名引数ではその名前のメソッドが返される．

名前付きの引数はメソッド定義で，あたかも `setRefClass()` への `methods` 引数中で提供されたかのようにクラス中にインストールされる． `setRefClass()` への呼び出しの代わりに，このようにメソッドを提供することはより明快なソースコードのために推奨される．詳細は節“参照メソッドを書く”を見よ．

クラスに対する全てのメソッドは，典型的にはパッケージの一部として，クラスを定義するソースコード中に定義されるべきである．特に，メソッドは名前空間を持つ付加されたパッケージ中で再定義出来ない：クラスメソッドはクラス定義のロックされた拘束をチェックする．

新しいメソッドは任意の現在定義されているメソッドを名前で参照できる `$methods()` へのこの呼び出し中で提供された他のメソッドを含む). しかしながら先に定義されたメソッドは再解析されないことはそれらが新しいメソッドを呼び出さないことを注意する (それが同じ名前の既存のメソッドを再定義しない限り).

メソッドを取り除くにはその新しい定義として `NULL` を与える．

`$fields()` 欄のリストを返す，各々は対応するクラスを持つ．定義中にアクセス関数が低供されている欄はクラス `"activeBindingFunction"` を持つ．

`$lock(...)` 引数中の欄名はロックされる；具体的に言うとロックメソッドが呼びだされた後に欄は一度設定出来るかもしれない．それを更に設定しようとする試みはエラーを惹き起こす．

もし引数なしに呼び出されると，メソッドはロックされた欄の名前を返す．

明示的なアクセス関数により定義された欄はロックできない (他方で，アクセス関数は引数無しで呼び出されるとエラーになるように定義できる).

欄をロックする全てのコードは通常クラスの定義の一部であるべきである；つまり，欄の読み出し専用特性はクラス定義の部分であり，後で加えられる動的な性質ではない．特に，欄は名前空間を持つ付加パッケージ中のクラス中ではロックできない：クラスメソッドはクラス定義のロックされた拘束をチェックする．ロックされた欄はその後でロック解除できない．

`$trace(what, ..., classMethod = FALSE)` このクラスから生成されたオブジェクトに対するメソッド `what` のトレース版を確立する．生成オブジェクトのトレースはクラスからのオブジェクトに対する `$trace()` メソッドと同様に動作するが，二つの違いを持つ．それはクラスオブジェクト中のメソッド定義自体を変更するため，トレースは全てのオブジェクトに適用され，トレースメソッドが適用されるものに対してだけではない．

次に，オプションの引数 `classMethod = TRUE` は生成オブジェクト自体のメソッドのトレースを許す．既定では， `what` はこのオブジェクトがそれに対する生成器であるクラス中のメソッド名の名前として解釈される．

`$accessors(...)` OOP プログラミングパラダイムを使う幾つかのシステムは名前による直接のアクセスではなく，各欄に対応する 取得と設定メソッドを推奨したり強要する．もしこのスタイルが好みで欄名 `abc` を `x$getAbc()` で取り出しそれを `x$setAbc(value)` で付値したければ， the `$accessors` メソッドが

指定された欄に対する取得と設定メソッドを作り出す便利な関数である．さもなければこの機構を使う理由はない．特に，それは “参照オブジェクト”節で説明された欄を関数で定義する一般的な能力とは何も関係がない．

実装

参照クラスはタイプ "environment" のデータ部を持つ S4 クラスとして実装されている．欄は環境中の名前付きオブジェクトに対応する．関数に関連する欄は active binding として実装されている．特に，指定されたクラスを持つ欄は欄への適正な付値を強制するアクティブな拘束の特殊な形式で実装されている．例えば data という欄は関連クラスからの任意のオブジェクトに対して形式 x$data の表現式で一般的にアクセスできる．このクラスへのメソッド中では欄は data という名前でアクセスできる．ロックされていない欄は x$data <- value の形式の表現式で設定できる．メソッドの内部では，欄は形式 x <<- value の表現式で付値出来る．演算子 非局所的付値 に注意しよう．この演算子の標準的な R の解釈はオブジェクトの環境中でそれを付値する．もし欄が定義されたアクセス関数を持てば，取得と設定はその関数を呼び出す．

メソッドがオブジェクトに対して適用されると，メソッドを定義する関数はオブジェクトの環境中にインストールされ，関数の環境は同じ環境である．

実装のせいで，新しい参照クラスは参照的なものと同じく非参照的な S4 クラスを継承できる．もし非参照クラスからのスロットが欄の代替物と考えられるならば，これは通常良いアイデアでは無い．ある種の特殊な引数が好まれるのではなければ，同じオブジェクトに対する関数と参照パラダイムの混在は概念的に不明瞭である．加えて，クラスに対する初期化メソッドはスロットからの欄をソートすべきで，このクラスのサブクラスに対する変則的な挙動を作り出す可能性が十分ある． S4 クラス中のスロットに類似した欄を定義し，それらをそのクラスからの S4 オブジェクトから初期化するのが一般的に好ましい．

システム間インタフェイス

幾つかの言語はクラスとクラスベースのメソッドを用いた参照ベースのプログラミングモデルを使う．用語と他の詳細の選択中の違いを別として，これらの言語の多くはここで解説されたプログラミングスタイルと互換性がある．言語への R インタフェイスが幾つかのパッケージ中に存在する．

ここでの参照クラス定義は R 中に現れる外部言語中のクラスに対するフックを提供する．クラス中の欄と/またはメソッドへのアクセスはインタフェイスを通じて利用可能にされるクラスに対応する R の参照クラスの定義により実装できる．典型的には，システム間インタフェイスは外部クラスの記述 (どのような欄とメソッドを持つか，欄に対するクラス，どれかが読み取り専用か，等) を与えて R クラスの生成の詳細の面倒を見る．特に，アクティブな拘束は取得と設定に対する欄のアクセスを許し，実際のアクセスはシステム間インタフェイスによりなされる．

R のメソッドと/または欄は任意の参照クラスに関してはクラス定義中に含めることが出来る．メソッドは欄を使ったり設定でき，そして欄やメソッドがインタフェイス由来か R 中で直接定義されたかによらず，他のメソッドを透過的に呼び出すことが出来る．

このアプローチを用いたシステム間インタフェイスについては，パッケージ Rcpp のバージョン 0.8.7 またはそれ以降を見よ．

デバッグ

標準の R のデバッグとトレース機能は参照メソッドに適用出来る．参照メソッドは debug とオブジェクトからのその類似物にそのオブジェクトに対する更なるメソッドの起動をデバッグするために渡すことが出来る；例えば debug(xx$edit)．

幾らかより柔軟な使い方が trace 関数の参照メソッドバージョンに対して利用できる．対応する $trace() 参照メソッドがオブジェクトか参照クラス生成器に対して利用できる (下の例の xx$trace() か mEdit$trace())． $trace() をオブジェクトに対して使うとそのオブジェクトに対する指定されたメソッドの将来の起動に対するトレースバージョンを

設定する． $trace() をそのクラスに対する生成器に対して使うとそのクラスからの全ての事後のオブジェクトに対するトレースバージョンを設定する(そしてある時は，もしメソッドが宣言されていなかったり前に起動されていなければ，そのクラスからの既存のオブジェクトに対しても).

どちらの場合も，参照クラスが S4 総称的ではないかもしれないので無意味な signature= を除けば標準の trace 関数への全ての引数が利用できる． これは対話的なデバッグに対する典型的なスタイルの trace(what, browser) と参照メソッドを対話的に編集する trace(what, edit = TRUE) を含む.

Author(s)

John Chambers

Examples

```
## 行列Bオブジェクトに対する簡単なエディタ.
## メソッド $edit() は値のある範囲を変更する：
## メソッド $undo() は最後の編集を下に戻す.
mEdit <- setRefClass("mEdit",
      fields = list( data = "matrix",
        edits = "list"),
      methods = list(
     edit = function(i, j, value) {
       ## 次の文字列はエディットメソッドをドキュメント化する
       'Replaces the range [i, j] of the
        object by value.
        '
         backup <-
             list(i, j, data[i,j])
         data[i,j] <<- value
         edits <<- c(edits, list(backup))
         invisible(value)
     },
     undo = function() {
       'Undoes the last edit() operation
        and update the edits field accordingly.
        '
         prev <- edits
         if(length(prev)) prev <- prev[[length(prev)]]
         else stop("No more edits to undo")
         edit(prev[[1]], prev[[2]], prev[[3]])
         ## trim the edits list
         length(edits) <<- length(edits) - 2
         invisible(prev)
     },
     show = function() {
       'Method for automatically printing matrix editors'
       cat("Reference matrix editor object of class",
          classLabel(class(.self)), "\n")
       cat("Data: \n")
       methods::show(data)
       cat("Undo list is of length", length(edits), "\n")
     }
     ))

xMat <- matrix(1:12,4,3)
```

```
xx <- mEdit(data = xMat)
xx$edit(2, 2, 0)
xx
xx$undo()
mEdit$help("undo")
stopifnot(all.equal(xx$data, xMat))

utils::str(xx) # 自明でないメソッドの欄と名前を示す

## 保存されたオブジェクトにメソッドを加える.
mEdit$methods(
     save = function(file) {
       'Save the current object on the file
        in R external object format.
       '
         base::save(.self, file = file)
     }
)

tf <- tempfile()
xx$save(tf)


## Not run:
## 参照クラスを継承する：行列のビューア
mv <- setRefClass("matrixViewer",
    fields = c("viewerDevice", "viewerFile"),
    contains = "mEdit",
    methods = list( view = function() {
        dd <- dev.cur(); dev.set(viewerDevice)
        devAskNewPage(FALSE)
        matplot(data, main = paste("After",length(edits),"edits"))
        dev.set(dd)},
        edit = # 先のメソッドを起動し，それから再プロット
          function(i, j, value) {
            callSuper(i, j, value)
            view()
          }))

## 初期化と終結化メソッド
mv$methods( initialize =
  function(file = "./matrixView.pdf", ...) {
    viewerFile <<- file
    pdf(viewerFile)
    viewerDevice <<- dev.cur()
    dev.set(dev.prev())
    callSuper(...)
  },
  finalize = function() {
    dev.off(viewerDevice)
  })

## オブジェクトのデバッグ：メソッド $edit() 中の browser() を呼び出す
xx$trace(edit, browser)

## メソッド $undo() 中のクラス mEdit からの全てのオブジェクトをデバッグ
mEdit$trace(undo, browser)
```

```
## End(Not run)
```

`removeMethod` *Remove a Method*

Description

Remove the method for a given function and signature. Obsolete for ordinary applications: Method definitions in a package should never need to remove methods and it's very bad practice to remove methods that were defined in other packages.

Usage

```
removeMethod(f, signature, where)
```

Arguments

`f, signature, where`
: As for `setMethod()`.

Value

`TRUE` if a method was found to be removed.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

`representation` 表現またはクラス定義に対するプロトタイプを作る

Description

これらはそれぞれスロットとスーパークラスを表現する用にデザインされたリストとプロトタイプ指定のリストを構成する旧式のユーティリティ関数である.

`setClass` に対する引数 `slots` と `contains` は今では推奨されないため `representation()` 関数は最早有用ではない.

`prototype()` 関数は対応する引数に対して以前使うことが出来るが, 同じ引数の単純なリストが同様に使える.

Usage

```
representation(...)
prototype(...)
```

Arguments

...	表現の呼び出しは単一の文字列である引数を取る．名前無しの引数は新しく定義されるクラスが拡張するクラスである；名前付き引数は新しいクラス中の明示的なスロットを命名し，各クラスがどのようなスロットを持つべきかを指定する． prototype の呼び出し中では，もし名前無しの引数があたえられると，それは無条件でプロトタイプオブジェクトに対する基礎を形成する．一つより多い名前無し引数を与えるのはエラーである．

Details

representation 関数は引数の妥当性に対するテストを適用する．各々はクラスの名前を指定しなければならない．

クラス名は representation が呼び出された時存在する必要はないが，もし存在すれば関数は継承クラスの各々から導入されたスロット名が重複していないかチェックする．

prototype への引数は普通スロットに対する名前付きの初期値，プラスオブジェクト自体を与えるオプションの第一引数である．名前無しの引数はもし定義に対するデータ部分がある時に典型的に有用である (下の例を見よ).

Value

representation の値は単に正当性をチェックされた後の引数のリストである．

prototype の値はプロトタイプとして使われるオブジェクトである．スロットは引数と一貫性があるように設定されているが，構成は中身の正当性をテストするのにクラス定義を使わない (プロトタイプオブジェクトは定義を作るために使われるので，これは不可能である).

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

setClass

Examples

```
## "numeric" オブジェクトであるべき直接に定義されたスロット
## "smooth" を持ち，クラス "track" を拡張する新しいクラスに対する表現
representation("track", smooth ="numeric")


setClass("Character",representation("character"))
setClass("TypedCharacter",representation("Character",type="character"),
          prototype(character(0),type="plain"))
ttt <- new("TypedCharacter", "foo", type = "character")


setClass("num1", representation(comment = "character"),
         contains = "numeric",
```

```
        prototype = prototype(pi, comment = "Start with pi"))
```

S3Part　　*S3* スタイルのオブジェクトと *S4* クラスのオブジェクト

Description

旧式(S3)のクラスは S4 クラスとして登録でき (setOldClass の呼び出しにより)，そして多くがそうされてきた．これらのクラスはそれから正規の S4 クラスに含められる (つまり，スーパークラス)ことが出来，形式的メソッドとスロットに S3 的挙動を与えることができる．関数 S3Part はそうしたオブジェクトの S3 部分を寄りだしたり置き換えたりする． S3Class は S3 スタイルのクラスを取り出したり置き換えたりする． S3Class はまた setClass への呼び出し中で S3methods=TRUE である S4 クラスからのオブジェクトに適用される．

下の詳細を見よ．また S3 と S4 間の強制変換が議論される；節 “S3 と S4 オブジェクト” を見よ．

Usage

```
S3Part(object, strictS3 = FALSE, S3Class)

S3Part(object, strictS3 = FALSE, needClass = ) <- value

S3Class(object)

S3Class(object) <-  value

isXS3Class(classDef)

slotsFromS3(object)
```

Arguments

object	登録された S3 クラスを拡張するあるクラスからのオブジェクトで，普通クラスがそのスーパークラスの一つとして setOldClass の呼び出しによって登録された S3 クラスを持つか，基本的なベクトル，行列または配列オブジェクトタイプを拡張するクラスからのものであるからである． 関数の殆どに対して S3 オブジェクトをそれがそれ自体の S3 部分であるという解釈で提供可能である．
strictS3	もし TRUE ならば，S3Part が返す値は S3 オブジェクトで，全ての S4 スロットは取り除かれる．さもなければ， S4 オブジェクトが常に返される；例えば，基礎にある S3 オブジェクトではなく S3 クラスのプロキシとしての setOldClass が作る S4 クラス．
S3Class	オブジェクト中の S3 スロットとして保管される character ベクトル．普通，そして既定では object からのスロットを保存する．
needClass	置き換え値はこのクラスかそのサブクラスであることが要求される．

value
: S3Part<- に対しては，オブジェクトの S3 部分に対する置き換え値．これは S4 オブジェクトである必要は無い；実際，これらのクラスからのオブジェクトを作る通常の方法は正しいクラスの S3 オブジェクトを new への引数として与えることによってである．

  S3Class<- に対しては，S3 メソッドの選択適用に対する代理として使われる文字列ベクトル．この置き換え関数は S3 のオブジェクト毎のメソッド選択に使うことが出来る．

classDef
: クラス定義オブジェクトで， getClass が返すようなもの．

Details

setOldClass への呼び出しで S3 クラスを登録したクラスは対応する文字列の S3 ベクトルを保持するスロット ".S3Class" を持つ．そうしたクラスのプロトタイプは setOldClass への引数で決められるこのスロットに対する値を持つ．

他の S4 クラスはもし引数 S3methods = TRUE が setClass に提供されると同じスロットを持つ；この場合スロットはクラスの S4 継承に設定される．

そうしたクラスを拡張する (含む)新しい S4 クラスは同様に同じスロットを持ち，既定でプロトタイプは setClass への contains= 引数で決定される値を持つ．

S4 クラスからの個別のオブジェクトはプロトタイプ中の値やその値の(S3)サブクラスに対応する S3 クラスを持つかもしれない．下の例を見よ．

strictS3 = TRUE である時の S3Part() は全ての形式的に定義されたスロットを消去しオブジェクトの S4 ビットをオフにして基礎にある S3 オブジェクトを構成する．strictS3 = FALSE の時は返されるオブジェクトは対応する S4 クラスのものである．一貫性と一般性のために， S3Part() は同様に基本ベクトル，行列，そして配列クラスを拡張するクラスに対しても動作する． R はそれが何を S3 クラスとして扱うかが少々曖昧なので ("ts" がそうであるが，"matrix" は違う)， S3Part() は S4 クラスが S3 か基本的オブジェクトタイプの適当なスーパークラスを持つ限り S3 (つまり非 S4)オブジェクトを返そうと試みる．

この一般性に基礎を置く一つの一般的な応用は S4 オブジェクトでないことが保証されるスーパークラスオブジェクトを得ることである．もし S4 オブジェクトに対してチェックを行うある関数を呼び出すと，閉じたループに陥らないように注意する必要がある (fooS4 は fooS3 を呼び出し，それは S4 オブジェクトに対してチェックを行いそして再び fooS4 を恐らく間接的に呼び出す)． S3Part() を strictS3 = TRUE と共に呼び出すことはそうしたループを避ける機構である．

S3 クラスのオブジェクトの内容は定義や保証を持たないので， S3 部分を含む計算はスロットの妥当性をチェック しない．R ではスロットは内部的には属性として実装されているので，それは S3 部分に存在するときはコピーされる．もし S3 クラスを拡張する S4 クラスが S4 スロット名として S3 属性の名前を使い，そして S3 コードが S4 定義に従う不正なクラスからのオブジェクトに属性を設定するときは深刻な問題が起こり得る．

しばしば S3Part は単にオブジェクトを希望のクラスに強制変換することで避けられるしそうすべきである；メソッドは as がスーパークラスを取り出すか置き換えるときにはスロットを正しく処理するように自動的に定義される．

関数 slotsFromS3() はオブジェクトの S3 部分に関連するスロットにアクセスするのに内部的に使われる総称的関数である．この関数に対するメソッドは setOldClass が S4Class 引数と共に呼び出される時自動的に作られる．普通唯一つの S3 スロットがあり，S3 クラスを含むが， S4Class 引数は S3 クラスが形式的 S4 スロットとして使うことが出来るある保証された属性を持つときは追加のスロットを提供できる． setOldClass のドキュメント中の対応する節を見よ．

Value

S3Part: S3 情報を返すか設定する (そして引数 S3Class と keepSlots に応じてもしかすると同様に S4 スロットも). 上の引数 strict の説明を見よ. もしそれが TRUE ならば返される値は S3 オブジェクトである.

S3Class: もしクラスが対応する .S3Class スロットを持てば, オブジェクト中に保管された S3 クラスの文字列ベクトルを返すか設定する. さもなければ現在関数の既定値は class.

isXS3Class: ClassDef により定義されたクラスが S3 クラスを拡張しているかどうかに応じて TRUE か FALSE を返す (特に, それが S3 クラスを保持するためのスロットを持つかどうか).

slotsFromS3: 関連するスロットのクラスのリスト, または任意の他のオブジェクトに対する空リスト.

S3 と S4 オブジェクト：変換機構

R 中のオブジェクトは, オブジェクトが S4 クラス由来として扱うかどうかを指示する内部ビットを持つ. このビットは isS4 によりテストされそして asS4 によりオン・オフ設定が出来る. 後者の関数はしかしながらチェックや解釈は行わない；全ての詳細が正しく処理されていることが明確に確実な時だけ使うべきである.

より使いやすい別法として, 仮想クラス "S3" と "S4" に強制変換するためのメソッドが定義されている. 表現式 as(object, "S3") と as(object, "S4") は S3 と S4 オブジェクトをそれぞれ返す. 加えてそれらは変換を適正な方法で行うように務め, S4 への強制の際はまた適正さをチェックする.

表現式 as(object, "S3") は二つの方法で使うことが出来る. 登録された S3 クラスの一つからのオブジェクトに対しては, 表現式がクラス属性が class(object) が含意する完全な多重文字列の S3 クラスであることを保証する. もし登録されたクラスが既知の属性/スロットを持てば, これらが又提供される.

as(object, "S3") の別の使用法は S4 オブジェクトを取りそしてそれを対応する属性を持つ S3 オブジェクトに変換する. もし S4 メソッドを起動することなしにオブジェクトを操作したければ, この変換は普通最も安全な方法である. これはデータ部分を持つ S4 クラスに対してのみ意味を持つ.

表現式 as(object, "S4") は class(object) の S4 定義からのオブジェクトを作り出すためにオブジェクト中の属性を使う. これは S3 計算による部分的に定義された S4 オブジェクトのバージョンを作り出す一般的な機構である (対応する引数を持つ new とたいして違わないが, しかしもし S4 オブジェクトが異なった引数を持つ初期化メソッドを持ったとしてもこの形式で使用できる).

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョン).

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョン.)

See Also

setOldClass

Examples

```
## S3 クラス "lm" を拡張する二つの例,
## クラス "xlm" は直接で, "ylm" は間接的
setClass("xlm", representation(eps = "numeric"), contains = "lm")
setClass("ylm", representation(header = "character"), contains = "xlm")

## stats::lm に対する例中で計算されるような lm.D9
y1 <- new("ylm", lm.D9, header = "test", eps = .1)
xx <- new("xlm", lm.D9, eps =.1)
y2 <- new("ylm", xx, header = "test")
stopifnot(inherits(y2, "lm"))
stopifnot(identical(y1, y2))
stopifnot(identical(S3Part(y1, strict = TRUE), lm.D9))

## これらのクラスは S3 部分として S3 サブクラスに挿入できる:
myData <- data.frame(time = 1:10, y = (1:10)^.5)
myLm <- lm(cbind(y, y^3)  ~ time, myData) # S3 class: c("mlm", "lm")
ym1 <- new("ylm", myLm, header = "Example", eps = 0.)

## "xlm" と "ylm" に対する同様のクラス,
## しかし S3 クラス class c("mlm", "lm") を拡張する
setClass("xmm", representation(eps = "numeric"), contains = "mlm")
setClass("ymm", representation(header="character"), contains = "xmm")

ym2 <- new("ymm", myLm, header = "Example2", eps = .001)

# しかしクラス "ymm" に対しては, クラス "lm" の S3 部分はエラー:
try(new("ymm", lm.D9, header = "Example2", eps = .001))


setClass("dataFrameD", representation(date = "Date"),
         contains = "data.frame")
myDD <- new("dataFrameD", myData, date = Sys.Date())

## S3Part() はデータ部分を持つ(.Data スロット)に適用される

setClass("NumX", contains="numeric", representation(id="character"))
nn <- new("NumX", 1:10, id="test")
stopifnot(identical(1:10, S3Part(nn, strict = TRUE)))

m1 <- cbind(group, weight)
setClass("MatX", contains = "matrix", representation(date = "Date"))
mx1 <- new("MatX", m1, date = Sys.Date())
stopifnot(identical(m1, S3Part(mx1, strict = TRUE)))
```

S4groupGeneric *S4 グループ総称的関数*

Description

スループ総称的関数に対してメソッドを定義できる. 各グループ総称的関数はそれに付随する幾つかのメンバー総称的関数を持つ.

グループ総称的関数に対して定義されたメソッドはグループの各メンバーに対して定義された同じメソッドをもたらすが，グループのメンバーに対して明示的に定義されたメソッドは同じシグネチャを持つグループ総称的なメソッドに優先する．

このドキュメント頁の中で紹介された関数は全て **methods** パッケージ中に在るが，この機構は setGroupGeneric を呼び出すことで任意のプログラマが利用できる (パッケージ **methods** が付加されている限り)．

Usage

```
## S4 グループ総称的関数：
Arith(e1, e2)
Compare(e1, e2)
Ops(e1, e2)
Logic(e1, e2)
Math(x)
Math2(x, digits)
Summary(x, ..., na.rm = FALSE)
Complex(z)
```

Arguments

x, z, e1, e2	オブジェクト．
digits	round や signif 中で使われる桁数．
...	メソッドへ・から渡される追加引数．
na.rm	論理値：欠損値を取り除くか?

Details

グループ総称的関数に対してはメソッドを通常のように setMethod への呼び出しで定義できる．グループ総称的関数は決して直接に定義されるべきではないことを注意する – もしそうなら適当なエラーメッセージが出る．グループ総称的関数に対するメタデータがロードされると，定義されたメソッドはグループのメンバーに対するメソッドになるが，同じシグネチャを持つメンバー関数に対して直接にメソッドが指定されていない時だけである．効果はグループ総称的定義は継承メソッドの前であるが直接指定されたメソッドの後に選択されるということである．メソッド選択に関するより多くについては Methods を見よ．

同様に対応する R オブジェクトが無い S3 グループの Math, Ops, Summary そして Complex がある，?S3groupGeneric を見よ．しかしこれらは S4 グループ総称的関数とは無関係である．

特定の総称的関数で定義されたグループのメンバーは getGroupMembers の呼び出しで得ることが出来る．現在このパッケージ中で定義されたグループ総称的関数に対してはメンバーは次の通りである：

Arith "+", "-", "*", "^", "%%", "%/%", "/"

Compare "==", ">", "<", "!=", "<=", ">="

Logic "&", "|".

Ops "Arith", "Compare", "Logic"

Math "abs", "sign", "sqrt", "ceiling", "floor", "trunc", "cummax", "cummin", "cumprod", "cumsum", "log", "log10", "log2", "log1p", "acos", "acosh", "asin", "asinh", "atan", "atanh", "exp", "expm1", "cos", "cosh", "cospi", "sin", "sinh", "sinpi", "tan", "tanh", "tanpi", "gamma", "lgamma", "digamma", "trigamma"

`Math2` "round", "signif"

`Summary` "max", "min", "range", "prod", "sum", "any", "all"

`Complex` "Arg", "Conj", "Im", "Mod", "Re"

`Ops` は単に三つのサブグループからなることを注意する．

これらのグループ中の全ての関数 (グループ総称的関数自体を除く)は R の基本関数である．これらは既定では S4 総称的関数ではなく，それらの多くはプリミティブ関数として定義されている．しかしながら，それでもそれらに対して個別にもグループ総称的経由でも形式的メソッドを定義できる．それは多かれ少なかれ期待されるように動作するが，背景では少々トリッキーであることは否めない．詳細は Methods を見よ．

`Math` グループの二つのメンバー，`log` そして `trunc` は追加引数として ... を持つことを注意する． `Math` に対するメソッドは唯一の形式的引数を持つため，それらを追加引数と共に呼び出すにはそれらに対して特別のメソッドを設定しなければならない．

グループ総称的関数に関する一層の詳細は *Software for Data Analysis* の節 10.5 を見よ．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して).

See Also

関数 `callGeneric` は総称的関数に対するメソッドを書くときにはほとんど常に関係する．下の例と *Software for Data Analysis* の節 10.5 を見よ．

S3 総称的関数に対しては S3groupGeneric を見よ．

Examples

```
setClass("testComplex", representation(zz = "complex"))
## "Complex" の全てのグループに対するメソッド
setMethod("Complex", "testComplex",
          function(z) c("groupMethod", callGeneric(z@zz)))
## Arg() に対する例外：
setMethod("Arg", "testComplex",
          function(z) c("ArgMethod", Arg(z@zz)))
z1 <- 1+2i
z2 <- new("testComplex", zz = z1)
stopifnot(identical(Mod(z2), c("groupMethod", Mod(z1))))
stopifnot(identical(Arg(z2), c("ArgMethod", Arg(z1))))
```

`SClassExtension-class` 継承(拡大)関係を表すクラス

Description

このクラスからのオブジェクトは単一の 'is' 関係を表す；これらのオブジェクトのリストは与えられたクラスの全ての拡張(スーパークラス)とサブクラスを表す．このオブジェクトは関係がどのように定義され，そして強制変換，検査また対応する置き換えられるかに関する情報を含む．

クラスからのオブジェクト

このクラスからのオブジェクトは直接の呼び出しから，そして setClass への呼び出し中の contains= 情報から，そして setClassUnion により作られた合併クラスから setIs で生成される．最後のケースでは情報は合併クラスの サブクラス定義に保管される (合併が封印クラスを含むことを許して).

スロット

subClass, superClass: 拡張されるクラス：setIs への from と and to 引数に対応する．

package: そのクラスが所属するパッケージ．

coerce: 関係により含意される as() 計算を実行する関数． これらの関数は直接使われるべきでは 無いことを注意する． それらは strict=TRUE を伴う as 関数だけを扱い，完全なメソッドがこれから機械的に作られる．

test: 関係が成立するかどうかをテストする関数． setIs へ明示的に指定された test に対する場合を除き，この関数は自明なものである．

replace: as(x, Class) <- value を実装するのに使われるメソッド．

simple: "logical" フラグで，もしこれが一つのクラスが別の定義中に含まれるか，またはクラスが仮想クラスを拡張すると明示的に述べられているという単純な関係であれば TRUE． 単純な拡張に対しては三つのメソッドが自動的に生成される．

by: もしこの関係が推移的に構成されていたならば，サブクラスからの最初の中間のクラス．

dataPart: "logical" フラグで，もし拡張クラスが実際にサブクラスのデータ部分ならば TRUE． この場合拡張クラスは基本クラスである (つまり，あるタイプ).

distance: 二つのクラス感の距離で，直接含まれるクラスに対しては 1 で，さもなければ間の世代数がプラスされる．

メソッド

シグネチャ中にクラス "SClassExtension" を持つメソッドは定義されていない．

See Also

is, as，そして classRepresentation クラス．

---

selectSuperClasses　クラスの(特定の種類の)スーパークラス

Description

ClassDef のスーパークラスを返す． 可能性として非仮想的だけであったり，または直接や単純なもの．

これらの関数は速くなるようにデザインされており，従って対応するクラス定義の contains スロットにだけ動作する．

Usage

```
selectSuperClasses(Class, dropVirtual = FALSE, namesOnly = TRUE,
                   directOnly = TRUE, simpleOnly = directOnly,
                   where = topenv(parent.frame()))

.selectSuperClasses(ext, dropVirtual = FALSE, namesOnly = TRUE,
                    directOnly = TRUE, simpleOnly = directOnly)
```

Arguments

Class	クラスの名前か(より効率的に)クラス定義 (getClass を見よ).
dropVirtual	非仮想的なスーパークラスだけが返されるべきかどうかを指示する論理値.
namesOnly	名前付きのクラス拡張のリストの代わりにベクトル名だけを返すかどうかを指示する論理値.
directOnly	直接のスーパークラスだけを返すべきかどうかを四s持する論理値.
simpleOnly	単純なクラス拡張だけを返すべきかどうかを指示する論理値.
where	(Class がクラス拡張で無い時だけ使われる) Class のクラス拡張が見つかる環境.
ext	.selectSuperClasses() に対してだけ，クラス拡張の list で，典型的には getClassDef(..)@contains.

Value

character ベクトル (既定としてもし namesOnly が真なら)またはクラス拡張のリスト (getClass の結果中の contains スロットとして).

Note

典型的なユーザレベルの関数は selectSuperClasses() であり .selectSuperClasses() を呼び出す；つまり後者は経験のある R ユーザが効率性のためだけに使われるべきである.

See Also

is, getClass；更により技術的なクラス classRepresentation のドキュメント.

Examples

```
setClass("Root")
setClass("Base", contains = "Root", representation(length = "integer"))
setClass("A", contains = "Base", representation(x = "numeric"))
setClass("B", contains = "Base", representation(y = "character"))
setClass("C", contains = c("A", "B"))

extends("C")   #-->  "C"  "A" "B"  "Base" "Root"
selectSuperClasses("C") # "A" "B"
selectSuperClasses("C", direct=FALSE) # "A" "B"  "Base"  "Root"
selectSuperClasses("C", dropVirt = TRUE, direct=FALSE)# 同上, "Root" 無し
```

setAs *Methods for Coercing an Object to a Class*

Description

A call to setAs defines a method for coercing an object of class from to class to. The methods will then be used by calls to as for objects with class from, including calls that replace part of the object.

Methods for this purpose work indirectly, by defining methods for function coerce. The coerce function is *not* to be called directly, and method selection uses class inheritance only on the first argument.

Usage

```
setAs(from, to, def, replace, where = topenv(parent.frame()))
```

Arguments

from, to	The classes between which the coerce methods def and replace perform coercion.
def	function of one argument. It will get an object from class from and had better return an object of class to. The convention is that the name of the argument is from; if another argument name is used, setAs will attempt to substitute from.
replace	if supplied, the function to use as a replacement method, when as is used on the left of an assignment. Should be a function of two arguments, from, value, although setAs will attempt to substitute if the arguments differ. *The remaining argument will not be used in standard applications.*
where	the position or environment in which to store the resulting methods. Do not use this argument when defining a method in a package. Only the default, the namespace of the package, should be used in normal situations.

Inheritance and Coercion

Objects from one class can turn into objects from another class either automatically or by an explicit call to the as function. Automatic conversion is special, and comes from the designer of one class of objects asserting that this class extends another class. The most common case is that one or more class names are supplied in the contains= argument to setClass, in which case the new class extends each of the earlier classes (in the usual terminology, the earlier classes are *superclasses* of the new class and it is a *subclass* of each of them).

This form of inheritance is called *simple* inheritance in R. See setClass for details. Inheritance can also be defined explicitly by a call to setIs. The two versions have slightly different implications for coerce methods. Simple inheritance implies that inherited slots behave identically in the subclass and the superclass. Whenever two classes are related by simple inheritance, corresponding coerce methods are defined for both direct and replacement use of as. In the case of simple inheritance, these methods do the obvious computation: they extract or replace the slots in the object that correspond to those in the superclass definition.

The implicitly defined coerce methods may be overridden by a call to setAs; note, however, that the implicit methods are defined for each subclass-superclass pair, so that you must override each of these explicitly, not rely on inheritance.

When inheritance is defined by a call to setIs, the coerce methods are provided explicitly, not generated automatically. Inheritance will apply (to the from argument, as described in the section below). You could also supply methods via setAs for non-inherited relationships, and now these also can be inherited.

For further on the distinction between simple and explicit inheritance, see setIs.

How Functions 'as' and 'setAs' Work

The function as turns object into an object of class Class. In doing so, it applies a "coerce method", using S4 classes and methods, but in a somewhat special way. Coerce methods are methods for the function coerce or, in the replacement case the function `coerce<-`. These functions have two arguments in method signatures, from and to, corresponding to the class of the object and the desired coerce class. These functions must not be called directly, but are used to store tables of methods for the use of as, directly and for replacements. In this section we will describe the direct case, but except where noted the replacement case works the same way, using `coerce<-` and the replace argument to setAs, rather than coerce and the def argument.

Assuming the object is not already of the desired class, as first looks for a method in the table of methods for the function coerce for the signature c(from = class(object), to = Class), in the same way method selection would do its initial lookup. To be precise, this means the table of both direct and inherited methods, but inheritance is used specially in this case (see below).

If no method is found, as looks for one. First, if either Class or class(object) is a superclass of the other, the class definition will contain the information needed to construct a coerce method. In the usual case that the subclass contains the superclass (i.e., has all its slots), the method is constructed either by extracting or replacing the inherited slots. Non-simple extensions (the result of a call to setIs) will usually contain explicit methods, though possibly not for replacement.

If no subclass/superclass relationship provides a method, as looks for an inherited method, but applying, inheritance for the argument from only, not for the argument to (if you think about it, you'll probably agree that you wouldn't want the result to be from some class other than the Class specified). Thus, selectMethod("coerce", sig, useInherited= c(from=TRUE, to= FALSE)) replicates the method selection used by as().

In nearly all cases the method found in this way will be cached in the table of coerce methods (the exception being subclass relationships with a test, which are legal but discouraged). So the detailed calculations should be done only on the first occurrence of a coerce from class(object) to Class.

Note that coerce is not a standard generic function. It is not intended to be called directly. To prevent accidentally caching an invalid inherited method, calls are routed to an equivalent call to as, and a warning is issued. Also, calls to selectMethod for this function may not represent the method that as will choose. You can only trust the result if the corresponding call to as has occurred previously in this session.

With this explanation as background, the function setAs does a fairly obvious computation: It constructs and sets a method for the function coerce with signature c(from, to), using the def argument to define the body of the method. The function supplied as def can have one argument (interpreted as an object to be coerced) or two arguments (the from object and the to class). Either way, setAs constructs a function of two arguments, with the second defaulting to the name of the to class. The method will be called from as with the object as the from argument and no to argument, with the default for this argument being the name of the intended to class, so the method can use this information in messages.

The direct version of the as function also has a strict= argument that defaults to TRUE. Calls during the evaluation of methods for other functions will set this argument to FALSE. The distinction is relevant when the object being coerced is from a simple subclass of the to class; if strict=FALSE

in this case, nothing need be done. For most user-written coerce methods, when the two classes have no subclass/superclass, the `strict=` argument is irrelevant.

The `replace` argument to `setAs` provides a method for `` `coerce<-` ``. As with all replacement methods, the last argument of the method must have the name `value` for the object on the right of the assignment. As with the `coerce` method, the first two arguments are `from, to`; there is no `strict=` option for the replace case.

The function `coerce` exists as a repository for such methods, to be selected as described above by the `as` function. Actually dispatching the methods using `standardGeneric` could produce incorrect inherited methods, by using inheritance on the `to` argument; as mentioned, this is not the logic used for `as`. To prevent selecting and caching invalid methods, calls to `coerce` are currently mapped into calls to `as`, with a warning message.

Basic Coercion Methods

Methods are pre-defined for coercing any object to one of the basic datatypes. For example, `as(x, "numeric")` uses the existing `as.numeric` function. These built-in methods can be listed by `showMethods("coerce")`.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

See Also

If you think of using `try(as(x, cl))`, consider `canCoerce(x, cl)` instead.

Examples

```
## using the definition of class "track" from \link{setClass}



setAs("track", "numeric", function(from) from@y)

t1 <- new("track", x=1:20, y=(1:20)^2)

as(t1, "numeric")

## The next example shows:
##  1. A virtual class to define setAs for several classes at once.
##  2. as() using inherited information

setClass("ca", slots = c(a = "character", id = "numeric"))

setClass("cb", slots = c(b = "character", id = "numeric"))

setClass("id")
setIs("ca", "id")
setIs("cb", "id")


setAs("id", "numeric", function(from) from@id)

CA <- new("ca", a = "A", id = 1)
CB <- new("cb", b = "B", id = 2)
```

```
setAs("cb", "ca", function(from, to )new(to, a=from@b, id = from@id))

as(CB, "numeric")
```

---

setClass　　クラス定義を作る

Description

表現(スロット) と/またはこれに含まれるクラス(スーパークラス), プラス他のオプションの詳細を指定しクラス定義を作る. 副作用として, クラス定義は指定された環境に保存される. 生成関数は setClass() の値として返され, もし非仮想的ならばクラスからオブジェクトを作るのに適している. 関数への多くの引数から Class, slots= そして contains= が普通必要とされる.

Usage

```
setClass(Class, representation, prototype, contains=character(),
         validity, access, where, version, sealed, package,
         S3methods = FALSE, slots)
```

Arguments

Class	クラスに対する文字列名.
slots	名前付きリストか名前付き文字列. 名前は新しいクラス中のスロット名で要素は対応するクラスの文字列名である. 同じ名前の二つのクラスは異なったパッケージから移入されるので稀なケースではスロットのクラスに関する曖昧さがあり, 対応する引数の要素は選択の曖昧さを無くすため "package" 属性を持たなければならない. 極端なケースとして名前無しの文字列を与えることが許され, 要素はスロット名として取られ, 全てのスロットは無制約のクラス "ANY" を持つ.
contains	このクラスのスーパークラスに対する名前(そしてオプションでパッケージスロット). 特別なスーパークラス "VIRTUAL" は新しいクラスが仮想的クラスとして生成されるようにする ; Classes 中の仮想的クラスに対する節を見よ.
prototype	このクラス中のスロットに対する既定データを提供するオブジェクト. もし与えられれば, prototype への呼び出しの使用はある種のチェックを行う.
where	その中に定義を保存する環境. 標準的な使用では与えるべきではない. パッケージに対するソースコード中に現れる setClass() への呼び出しに対しては既定値はパッケージの名前空間. タイプされるかセッションのトップレベルでソース読み込みされた呼び出しに対しては既定値は大局的環境.

validity もし提供されると，このクラスからのオブジェクトに対する適正度チェックメソッド (もしその引数がこのクラスに対する適正なオブジェクトならば TRUE を返し，さもなければ失敗を説明する一つまたはそれ以上の文字列)であるべきである．詳細は validObject を見よ．

S3methods, representation, access, version
これら全ての引数は R のバージョン 3.0.0 から廃止予定とされ 避けるべきである．

S3methods は旧式スタイルのメソッドがこのクラスを含んで書かれることを指示するフラグ． R の新しいバージョンは形式的と旧式スタイルのメソッドに一貫性があるようにマッチすることを試みるので，この引数はあまり問題ではない．

representation は slots と contains の双方に含まれる S から継承される引数であるが，後者の二つの引数の使用はより明快で推奨される．

access と version は S-Plus との歴史的な互換性のために含まれているが無視される．

sealed もし TRUE ならばクラス定義は封印されるので， setClass への別の呼び出しはこのクラス名に対しては失敗する．

package このクラスに対するオプションのパッケージ名．非常に稀にしか使われるべきではない．既定ではクラス定義がその中に付値されているパッケージの名前．

Value

クラスからのオブジェクトに対して適当な生成関数を不可視で返す．この関数の呼び出しはクラスに対する new の呼び出しを生成する．呼び出しは任意個数の引数を取り，それらは初期化メソッドに渡される．もしクラスかそのスーパークラスの一つに対する initialize メソッドが与えられないと，既定メソッドはスロットの一つの名前を持つ名前付き引数を期待する．

プログラミングの分かりやすさから，典型的には生成関数はクラスの名前を付けられる．これは要求では無く，そしてクラスからのオブジェクトはまた new から直接に生成することも出来る．生成関数の利点は少しより単純で明快な呼び出しであり，呼び出しがクラスのパッケージ名を含むことである (もし異なったパッケージからの二つのクラスが同じ名前を持てば曖昧さが無くせる)．

もしクラスが仮想的ならば，生成器か new() のどちらかからのオブジェクトを生成しようとするとエラーになる．

基本的な用法：スロットと継承

クラス名以外の二つの本質的な引数は slots と contains で，明示的なスロットと継承(スーパークラス)を定義する．これらの引数は併せてこのクラスからの全ての情報を定義する；つまり，それらの各々に対して必要とされる全てのスロットとクラスの名前である．

クラス名はどのメソッドがこのクラスからのオブジェクトに直接適用されるかを決定する．継承情報はどのメソッドが継承により間接的に適用されるかを指定する． Methods を見よ．

クラス定義中のスロットは slots により直接指定される全てのスロットと全ての含まれるクラス中のスロットの合併である．与えられた名前のスロットは一つしか許されない；特に直接と継承されたスロット名はユニークでなければならない．しかしながら，これは同じクラスが一つ以上のパスで継承されることを妨げない．

contains= 引数中の一種類の要素は特殊で，一つの R オブジェクトタイプかいくつかの他の特殊な R タイプ(matrix と array)の一つを指定する．下のオブジェクトタイプからの継承の節を見よ．

スロット名 "class" と "Class" は許されない．特殊な意味を持つ他のスロット名がある；これらの名前は "." 文字で始まる．安全のために，自分自身の全てのスロットをあるアルファベット文字で始まるように定義すべきである．

オブジェクトタイプからの継承

他の S4 クラスを含む以外に，クラス定義は S3 クラス(次の節を見よ)か組み込みの R の擬似クラス— R のオブジェクトタイプの一つか特殊な R の擬似クラス "matrix" と "array"—を含むことが出来る．一つのクラスは他のタイプを最大一つ直接的または間接的に含むことが出来る．もしそうであれば，その含まれたクラスはクラスの “データ部分”を決定する．

新しいクラスからのオブジェクトは含まれるタイプの組み込みの挙動を継承使用と試みる．ベクトル，関数そして表現式を含む正規の R データタイプの場合，実装は相対的に直接的である．クラスからの任意のオブジェクトに対して， typeof(x) は含まれる基本的タイプである；そして特殊と擬似スロット，.Data，は対応するクラスとして表示される．下の例 "numWithId" を見よ．

クラスはまた "vector", "matrix" または "array" を継承できる．これらのオブジェクトのデータ部分は任意のベクトルデータタイプであり得る．

これらのタイプやクラスを含ま 無い任意のクラスからのオブジェクトに対しては，typeof(x) は "S4" になる．

ある種の R データタイプは，それらが非局所参照であるか複製されない他のオブジェクトであるという意味で普通の挙動をしない．例はクラス "environment", "externalptr" そして "name" に対応するそれらを含む．これらは，属性の設定があらゆる文脈でオブジェクトを上書きするために，ユーザ定義のクラス (S4 でも S3 でも)を持つオブジェクトに対するタイプにはなれない．そうしたタイプを継承するクラスを定義することは継承オブジェクトを予約スロットに補完する間接的な機構を通じて可能である．下のクラス "stampedEnv" に対するクラスの例を見よ． S3 メソッドの選択適用と関連する as.*type*() 関数は正しく動作すべきであるが，オブジェクトのタイプを直接使うコードはしない．

又低水準の計算に渡されるオブジェクトクラス中に定義された任意のスロット無しの基底にあるオブジェクトタイプであることを留意せよ．完全な情報を返すためには，普通データパートを設定するメソッドを書かねばならない．

S3 クラスからの継承

旧式のスタイルの S3 クラスは形式的定義を持たない．オブジェクトはそれらのクラス属性がクラスメイト考えられる文字列を含む時そのクラス“由来”とされる．

そうしたクラスを形式的クラスとメソッドと共に使うことは，オブジェクトの中身や継承メソッドの一貫性関する保証が無いため危険な仕事になる．その上で， S3 クラスを継承するクラスを定義することは，クラスが旧式クラスとして登録されていれば (setOldClass を見よ)依然可能である．

大まかに言えば，S3 と S4 メソッドの選択適用はどちらのシステム中の継承に関して意味があるように振る舞うように務める．S4 オブジェクトをあたえた時，S3 メソッドの選択適用と inherits 関数は S4 継承情報を使うべきである． S3 オブジェクトをあたえた時，S4 総称的関数は，継承が setOldClass を使って宣言されている限り， S3 継承を用いて S4 メソッドを選択適用する．

クラスとパッケージ

クラス定義は普通パッケージに属する (しかし同様に大局的環境中でコマンドラインの表現式の評価やコマンドラインからソース読み込みされるファイル中で定義可能である). 対応するパッケージ名はクラス定義の一部分である；つまりその定義を保持する classRepresentation オブジェクトの一部分である. このように, 同じ名前の二つのクラスが異なったパッケージ中に存在できる.

クラス名が setClass への呼び出し中のスロットやスーパークラスに対して提供されると, 曖昧さを避けるためにそうあるべきなように, 問題の呼び出しがパッケージに対するソース中に直接現れることを仮定して, 対応するクラス定義が現在のパッケージの名前空間から探される. クラス定義は現在の名前空間中, その名前空間空間に対する移入物, またはメソッドパッケージで定義された基本クラス中, で見つかるべきである. (メソッドパッケージは "CMD check" ユーティリティがこれらのクラスを見つけられるようにパッケージの "DESCRIPTION" ファイル中の Depends ダイレクティブ中に含められるべきである.)

この規則がクラスをユニークに特定出来ない (一つ以上の移入されたパッケージ中に現れるため)時は文字列名の packageSlot がその名前で提供される必要がある. これは滅多に起きるべきではない.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して.)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して.)

See Also

クラスの一般的な議論は Classes, メソッドの同様の議論は makeClassRepresentation, Methods.

Examples

```
## 二つのスロットを持つ簡単な例
track <- setClass("track", slots = c(x="numeric", y="numeric"))
## クラスからのオブジェクト
t1 <- track(x = 1:10, y = 1:10 + rnorm(10))

## 先を拡張するクラス, スロットをもうひとつ加える
trackCurve <- setClass("trackCurve",
slots = c(smooth = "numeric"),
contains = "track")

## スーパークラスオブジェクトを含むオブジェクト
t1s <- trackCurve(t1, smooth = 1:10)

## "trackCurve" に類似のクラスだが,
## "y" に行列と "smooth" スロットを許す異なった構造を持つ
setClass("trackMultiCurve",
         slots = c(x="numeric", y="matrix", smooth="matrix"),
         prototype = list(x=numeric(), y=matrix(0,0,0),
                          smooth= matrix(0,0,0)))
## これらのクラスを使う更なる例に対しては ?setIs を見よ
```

```
## 組み込みのデータタイプ "numeric" を拡張するクラス

numWithId <- setClass("numWithId", slots = c(id = "character"),
         contains = "numeric")

numWithId(1:3, id = "An Example")

## タイプ "environment" の参照クラスを継承
stampedEnv <- setClass("stampedEnv", contains = "environment",
                       slots = c(update = "POSIXct"))
setMethod("[[<-", c("stampedEnv", "character", "missing"),
   function(x, i, j, ..., value) {
       ev <- as(x, "environment")
       ev[[i]] <- value  # 環境中のオブジェクトを更新
       x@update <- Sys.time() #そして更新日時
       x})


e1 <- stampedEnv(update = Sys.time())

e1[["noise"]] <- rnorm(10)
```

---

setClassUnion　　他のクラスの合併として定義されるクラス

Description

クラスは他のクラスの 合併として定義出来る；つまり幾つかの他のクラスのスーパークラスとして定義された仮想クラスとして．クラス合併は，幾つかのクラスのどれかを提供することを許したいときは，メソッドのシグネチャ中や他のクラス中のスロットとして有用である．

Usage

```
setClassUnion(name, members, where)
isClassUnion(Class)
```

Arguments

name	新しい合併クラスの名前.
members	この合併のメンバーであるべきクラス.
where	クラス定義を保存する場所；既定では setClassUnion 呼び出しが登場するパッケージの環境，またはもしパッケージソース外から呼び出された場合は大局的環境.
Class	クラスの名前または定義.

Details

members 中のクラスは合併を作る前に定義されていなければならない．しかしながら，下の例のようにメンバーは既存の合併に後から付け加えることが出来る．クラス合併は他のクラス合併のメンバーになり得る．

クラス合併の定義中のプロトタイプはもし "NULL" が合併のメンバーならば NULL であり，そしてさもなければ最初のメンバーのクラスのプロトタイプオブジェクトである (バージョン 2.15.0 の R に関しては；初期のバージョンはもしそれが不正であれば NULL プロトタイプを持っていた).

クラス合併はその定義が封印されているクラス (例えば基本データタイプまたは R の基本やメソッドパッケージ中で定義された他のクラス) により拡張されているクラスを作る唯一の方法である． "other" がクラス合併でない限り setIs("function", "other") と述べることは出来ない．一般に，この形式の呼び出し setIs は述べられた最初のクラスの定義を変更する ("other" を "function" の定義中に含まれるスーパークラスのリストに加える).

クラス合併はこれを最初のクラス定義を変更すること無く，代わりにクラス合併のサブクラススロット中の情報を保存することにより処理する，このテクニックが動作するためには， extends(class1, class2) のような表現式に対する内部計算は通常のクラスに対するのとは異なった動作をする；特にそれらは任意のクラスが class1 のスーパークラスと class2 のサブクラス間で共通であるかどうかをテストする．

クラス合併に対するクラス定義オブジェクトはそれ自体 classRepresentation の拡張であるクラス "ClassUnionRepresentation" という特殊なクラスを持つため，クラス合併に対する異なった挙動が可能にされている．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

Examples

```
## 数値か論理値データのどちらかに対するクラス
setClassUnion("maybeNumber", c("numeric", "logical"))

## 別のクラスのデータ部分として合併を使う
setClass("withId", representation("maybeNumber", id = "character"))

w1 <- new("withId", 1:10, id = "test 1")
w2 <- new("withId", sqrt(w1)%%1 < .01, id = "Perfect squares")

## 合併 "maybeNumber" にクラス "complex" を加える
setIs("complex", "maybeNumber")

w3 <- new("withId", complex(real = 1:10, imaginary = sqrt(1:10)))

## 既存の合併 "OptionalFunction" を含むクラス合併
setClassUnion("maybeCode",
    c("expression", "language", "OptionalFunction"))

is(quote(sqrt(1:10)), "maybeCode")  ## TRUE
```

`setGeneric` 新しい総称的関数を定義する

Description

与えられた名前の新しい総称的関数を作る，つまりこの関数に対して定義された形式的メソッドの間から引数のクラスに従ってメソッドを選択適用する関数.

Usage

```
setGeneric(name, def= , group=list(), valueClass=character(),
           where= , package= , signature= , useAsDefault= ,
           genericFunction= , simpleInheritanceOnly = )

setGroupGeneric(name, def= , group=list(), valueClass=character(),
                knownMembers=list(), package= , where= )
```

Arguments

`name`	総称的関数の文字列名．最も簡単な(そして推奨される)呼び出し `setGeneric(name)` はこの名前の関数を探し，そしてもし見つかった関数が総称的でなければ対応する総称的関数を作る，後者の場合，存在する関数は既定メソッドになる.
`def`	総称的関数関数を定義するオプションの関数オブジェクト．もし既存の非総称的関数を総称的にしたければこの引数を与えてはならない．この場合普通引数が一つの単純な呼び出しを使いたいであろう. もしこの名前の関数が無いか，ある理由からこの関数を総称的関数の定義に使いたくなければ `def` を与えない．この場合形式的引数と総称的関数に対する既定値は `def` から取られる．殆どの場合， `def` の本体はそうすると既存関数が引数一つの呼び出し中で行うように既定メソッドを定義する. もし既定メソッドを 持たない新しい総称的関数を作りたければ， `def` は `name` と同じ文字列を持つ `standardGeneric` の呼び出しだけであるべきである.
`group`	オプションで，この関数が属するグループ総称的関数の名前を与える文字列．メソッド選択中のグループ総称的関数の詳細については Methods を見よ.
`valueClass`	一つまたはそれ以上のクラス名のオプションのの文字列ベクトル．総称的関数が返す値はこのクラスを持つ(または拡張する)，またはクラスの一つ；さもなければエラーが起きる.
`package`	この関数が関連付けられているパッケージの名前．普通自動的に決定される (もしあればパッケージは非総称的バージョンを含むから，さもなければこの総称的関数が保存されるパッケージ).
`where`	結果の初期メソッドともしかすると総称的関数をどこに保存するか；既定ではトップレベルの環境中に保存する.
`signature`	オプションで，`setMethod` への呼び出し中で，この関数に対するメソッドのシグネチャ中に現れることが出来る関数への形式的引数中からの名前のベクトル．もし `...` が形式的引数の一つならば，それは特別に扱われる．バージョン 2.8.0 の R 以来，`...` は総称的関数のシグ

ネチャでも良くなった．そうするとメソッドはもしそれらのシグネチャが全ての ... 引数にマッチした時に選択される．詳細はトピック dotsMethods に対するドキュメントを見よ．現在のバージョンでは，... と他の引数をシグネチャ中で混ぜ合せることは不可能である (この制約は将来のバージョンでは外されるかもしれない).

既定では，シグネチャは非総称的関数に対応する暗黙の総称的関数から推測される．もし非総称的関数が定義されていないと，既定は ... を除く全ての形式的引数で，関数定義に登場する順序である．... が唯一の形式的引数の場合，それはまた既定のシグネチャである． ... をある他の引数を持つ関数中のシグネチャとして使うためには，シグネチャ引数を明示的に与えなければならない．より詳細は下の“暗黙の総称的関数”節を見よ．

useAsDefault
: 既定引数の選択を上書きする．引数 useAsDefault を既定として使われる関数か論理値として提供できる．この引数は今では滅多に必要でない．‘詳細’節を見よ．

simpleInheritanceOnly
: 選ばれたメソッドが単純な継承だけを通じて継承されることを要求するためにはこの引数を TRUE として与える；つまり， setClass への contains= 引数中で指定されたスーパークラスから，またはクラス合併または他の仮想的クラスから．総称的関数は，もしそれが変換されたものでなく完全なオリジナルのオブジェクトを得ることが保証される必要があれば，総称的関数は単純な継承を要求すべきである．単純な継承を必要とする関数の例は initialize である，定義からそれはその引数と同じクラスからのオブジェクトを返さなければならないからである．

そして show であり，それはその引数としてオブジェクトの完全な記述を与えることを要求するためである．

genericFunction
: 使わないこと；(可能性として)内部使用専用．

knownMembers
: (setGroupGeneric に対してだけ．) このグループのメンバーに対して知られている関数の名前．この情報はグループ総称的関数に関する情報が変更された時メンバーの総称的関数のキャッシュされた定義をリセットするのに使われる．

Value

setGeneric 関数はその副作用のために存在する：後で指定されるメソッドを許す総称的関数を保存する． name が返される．

基本的使用法

setGeneric 関数はその関数に対してあるメソッド定義を用意するために総称的関数を初期化するために呼び出される．

最も単純で最も普通の状況は name はすでに通常の非総称的で非プリミティブな関数で，この関数を総称的関数に変えたいというものである．この場合，最もしばしば name だけを提供する，例えば：

```
setGeneric("colSums")
```

この名前の既存の関数がある付加されたパッケージ(この場合パッケージ "base") になければならない．この関数の総称的バージョンは現在のパッケージ中に作られる(またはもし setGeneric() への呼び出しが普通のソースファイルからかコマンドラインから入力されたならば大局的環境)．既存の関数は既定メソッドになり，そして新しい総称的関数の

パッケージスロットはオリジナルの関数の位置に設定される (例では "base"). 同じ総称的関数定義が毎回作られることは重要な特性である. 例では print の定義とそれがどこで見つかるかにだけ依存する. 総称的関数の signature は既定では, 形式的引数のどれが指定されたメソッド中で使うことが出来るかを定義する ... を除く形式的引数全てに設定される.

この形式で setGeneric() を呼び出すことは厳密には同じ関数に対して setMethod() を呼び出す前には不要であることを注意する. もし総称的でなければ, setMethod は setGeneric 自体の呼び出しを実行する. 関数を総称的にしたいと明示的に宣言することはより良いプログラミングスタイルと考えられる ; しかしながら結果中の唯一の違いはもしそうしなければ総称的関数の生成を注意するメッセージを作り出す.

ベースパッケージ中のプリミティブ関数の明示的な総称的バージョンを作ることは出来ない(し必要もない). 総称的関数として扱うことが出来るものは, 効率性の配慮を満たすために, 内部的なコードからメソッドが選択し適用され, そしてその他は総称的に出来ない. 下のプリミティブ関数の節を見よ.

また内部的に選択適用を行う非プリミティブ関数の明示的な総称的バージョンを作ることは不必要である. これらは unlist と as.vector を含む.

上の解説は非総称的関数を持つパッケージが暗黙の総称的バージョンを創りだしていない時の効果である. さもなければ, 使われるのはこの暗黙の総称的関数である. 下の暗黙の総称的関数の節を見よ. どちらの場合も, 本質的な結果は総称的関数の 同じバージョンが毎回作られるということである.

setGeneric() の二つ目の通常の用法は任意の既存の関数とは無関係に新しい総称的関数を作り, しばしば既定メソッドを持たない. この場合, 関数に対する引数を定義するために関数定義の骨格を与える必要がある. 総称的関数の本体は普通標準の形式 standardGeneric(name) であり, ここで name は総称的関数の引用符付きの名前である. setGeneric をこの形式で呼び出す時は, 普通 def 引数をこの形式の関数として提供する. 下の二番目と三番目の例を見よ.

引数 useAsDefault は新しい総称的関数の既定メソッドを制御する. もし別の指定がなければ, setGeneric は既定として使うために関数の非総称的バージョンを探す. 従って, もし適当な既定メソッドがなければ, 先ずこれを非総称的関数として設定し, それからこの節の最初にある setGeneric を引数一つで呼び出すのがしばしばより簡単である. 下の例の節の最初の例を見よ.

もし既存の関数を既定としたくなければ, useAsDefault 引数を提供する. この引数は既定メソッドにしたい関数で良く, 既定無しにしたければ FALSE する (つまり関数の呼び出しに対して選択される直接か継承メソッドがもしなければエラーを起こす).

詳細

setGeneric() の呼び出しの大多数は既存の関数がメソッドも持つことが出来ることを保証する一つの引数持つか, 新しい総称的関数とオプションで既定メソッドを作るための引数 name と def を持つ. もしそれが希望のものでなければ先に読み進んで欲しい.

もし既存の挙動を変更 (典型的には別のパッケージ中のもの)したければ, 対応して setGeneric に引数を提供する. どのような変更がされようと, 新しい総称的関数は 現在のパッケージに設定されたパッケージスロット付きで付値される. このステップは, 作成中のバージョンが最早他のパッケージ中の関数とは異なるので必要とされる. これが起きたことを知らせるメッセージがプリントされ, 二つの関数間の違いの一つを注意する. 二つのバージョンが今やメソッドに対して競合し, プログラミング中のミスの多くの可能性を持つため, これは悪いアイデアになりがちである.

総称的関数の本体は普通 standardGeneric の呼び出しによるメソッドの選択適用を除いて何も行わない. ある種の状況下では総称的関数自体中である追加の計算を行いたいかもしれない. 自分の関数が結局 standardGeneric を呼び出す限りこれは許される (しか

しながら恐らく良いアイデアでは無い関数のより簡単には理解できない挙動を招くという意味で). もし自分の総称的関数の明示的な定義が standardGeneric を呼びださなければ，この関数に対するどのメソッドも未だ選択適用されていないために問題が起きる

既定では，総称的関数は如何なるオブジェクトも返すことが出来る．もし valueClass が提供されると，それはクラス名のベクトルであるべきである；メソッドが返す値はそうすると指定されたクラスの一つに対して is(object, Class) を満たすことが要求される．空(つまり長さがゼロ)のベクトルは何も許されないことを意味する．結果に対するより複雑な要求は非標準的な総称的関数を定義することで明示的に指定できる．

setGroupGeneric 関数はそれがグループ総称的関数を作ることを除いて setGeneric と似たような挙動をするが， 2つの点で通常の総称的関数と異なる．先ずこの関数は直接呼び出すことは出来ず，作られた関数の本体はこの情報を持つ停止呼び出しを含む．次にグループ総称的関数は，検索リスト中の変更やメソッドの直接的な指定等で，グループ定義が変更された時にメンバーを更新し続けるために，グループの既知のメンバーに関する情報を含む．

暗黙の総称的関数

非総称的関数を“総称的に変換する”ということはより正確には関数が対応する暗黙の総称的関数に変換されることを意味する．もし特別なアクションが取られないと，任意の関数は暗黙のうちに同じ引数の総称的関数に対応し，... 以外の引数を使うことが出来る．この総称的関数のシグネチャは ... を除いて順番に形式的引数のベクトルである．

パッケージに対するソースコードはパッケージ中の任意の関数の暗黙の総称的関数バージョンを定義できる(機構については implicitGeneric を見よ)．一般に他人のパッケージ中に暗黙の総称的関数を出来ない．暗黙の総称的関数を定義する普通の理由はある引数がシグネチャ中に登場することを妨げることにあり，もし引数が文字通りの意味で使いたいかまたは何らかの理由で遅延評価を強制したければそうしなければならない．暗黙の総称的関数はまた予め定義しておきたいあるメソッドを含むことが出来る；実際，総称的関数は任意の非総称的関数の総称的関数であり得る．暗黙の総称機構はまた総称バージョンを禁止することに使える(prohibitGeneric を見よ).

定義されたにせよ自動的に推測されたにせよ，暗黙の総称的関数は暗黙の総称的関数が別のパッケージ中にある時は setGeneric が作った総称的関数と比較される．もし二つの関数が同一であれば，作られた総称的関数の package スロットは暗黙の総称的関数を含むパッケージの名前を持つ．さもなければ，スロットは総称的関数が付値されるパッケージの名前になる．

この規則の目的は総称的関数とパッケージ名の特定の組み合わせに対して定義された全てのメソッドが単一で一貫性のある総称的関数のバージョンに対応することを保証することにある． setGeneric を引数として name だけと可能性として package とともに呼び出すことはもし存在すれば暗黙の総称的関数を得ることを保証する．

他の引数のどれかを含むことは総称的関数の新しい局所的なバージョンを強制する．もし新しいバージョンを作りたくなければ，追加の引数を使わない．

総称的関数とプリミティブな関数

幾つかの基本的な R 関数はプリミティブ関数として特殊な実装を持ち， R の言語定義の評価ではなく，基底にある C コードで直接に評価される．殆どは暗黙の総称性を持ち(implicitGeneric を見よ)，そしてメソッドがそれらに対して定義される(グループメソッドを含む)や否や総称的関数になる．他は総称的関数に出来ない．

そうした関数にメソッドが定義された時も， C バージョンが呼び出され続けるために総称的バージョンは検索リスト上で不可視である．メソッド選択は C コード中で初期化される．しかしながら，結果はメソッドをプリミティブ関数をシグネチャ中のクラスの少なくとも一つが形式的 S4 クラスであるシグネチャに制限する．

プリミティブ関数の総称的バージョンを見るには， getGeneric(name) を使う．関数 isGeneric は現在のセッション中の関数に対してメソッドが定義されているかどうかを知らせる．

S4 メソッドは ‘internal generic’ であるようなプリミティブ関数と，プラス %*% に対してだけ設定できることを注意する．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

一般的議論は Methods とそこでのリンク， ... に関して選択適用を行うメソッドに対しては dotsMethods，そしてメソッド定義については setMethod.

Examples

```
## 新しい総称的関数を既定メソッドで作る
setGeneric("props", function(object) attributes(object))

## 既定メソッド無しの新しい総称的関数
setGeneric("increment",
  function(object, step, ...)
    standardGeneric("increment")
)


### 非標準的な総称的関数．メソッドが空でない文字列を返すことを要求する
### (setGeneric 呼び出し中の valueClass = "character" よりも強い要求)

setGeneric("authorNames",
    function(text) {
      value <- standardGeneric("authorNames")
      if(!(is(value, "character") && any(nchar(value)>0)))
        stop("authorNames methods must return non-empty strings")
      value
      })



## グループ総称的関数の例，クラス "track" を用いる；
## その定義については 'setClass' のドキュメントを見よ

## Arith グループに対するメソッドの定義

setMethod("Arith", c("track", "numeric"),
 function(e1, e2) {
  e1@y <- callGeneric(e1@y , e2)
  e1
})
```

```
setMethod("Arith", c("numeric", "track"),
 function(e1, e2) {
  e2@y <- callGeneric(e1, e2@y)
  e2
})

## 今や算術演算はメソッドを選択適用する：

t1 <- new("track", x=1:10, y=sort(stats::rnorm(10)))

t1 - 100
1/t1
```

`setGroupGeneric` *Create a Group Generic Version of a Function*

Description

The `setGroupGeneric` function behaves like `setGeneric` except that it constructs a group generic function, differing in two ways from an ordinary generic function. First, this function cannot be called directly, and the body of the function created will contain a stop call with this information. Second, the group generic function contains information about the known members of the group, used to keep the members up to date when the group definition changes, through changes in the search list or direct specification of methods, etc.

All members of the group must have the identical argument list.

Usage

```
setGroupGeneric(name, def= , group=list(), valueClass=character(),
                knownMembers=list(), package= , where= )
```

Arguments

`name`	the character string name of the generic function.
`def`	A function object. There isn't likely to be an existing nongeneric of this name, so some function needs to be supplied. Any known member or other function with the same argument list will do, because the group generic cannot be called directly.
`group, valueClass`	arguments to pass to `setGeneric`.
`knownMembers`	the names of functions that are known to be members of this group. This information is used to reset cached definitions of the member generics when information about the group generic is changed.
`package, where`	passed to `setGeneric`, but obsolete and to be avoided.

Value

The `setGroupGeneric` function exists for its side effect: saving the generic function to allow methods to be specified later. It returns `name`.

References

Chambers, John M. (2016) *Extending R* Chapman & Hall

See Also

Methods_Details and the links there for a general discussion, dotsMethods for methods that dispatch on ..., and setMethod for method definitions.

Examples

```
## Not run:
## the definition of the "Logic" group generic in the methods package
setGroupGeneric("Logic", function(e1, e2) NULL,
    knownMembers = c("&", "|"))

## End(Not run)
```

---

setIs *Specify a Superclass Explicitly*

Description

setIs is an explicit alternative to the contains= argument to setClass. It is only needed to create relations with explicit test or coercion. These have not proved to be of much practical value, so this function should not likely be needed in applications.

Where the programming goal is to define methods for transforming one class of objects to another, it is usually better practice to call setAs(), which requires the transformations to be done explicitly.

Usage

```
setIs(class1, class2, test=NULL, coerce=NULL, replace=NULL,
      by = character(), where = topenv(parent.frame()), classDef =,
      extensionObject = NULL, doComplete = TRUE)
```

Arguments

class1, class2	the names of the classes between which is relations are to be examined defined, or (more efficiently) the class definition objects for the classes.
coerce, replace	functions optionally supplied to coerce the object to class2, and to alter the object so that is(object, class2) is identical to value. See the details section below.
test	a *conditional* relationship is defined by supplying this function. Conditional relations are discouraged and are not included in selecting methods. See the details section below. The remaining arguments are for internal use and/or usually omitted.
extensionObject	alternative to the test, coerce, replace, by arguments; an object from class SClassExtension describing the relation. (Used in internal calls.)

doComplete	when TRUE, the class definitions will be augmented with indirect relations as well. (Used in internal calls.)
by	In a call to setIs, the name of an intermediary class. Coercion will proceed by first coercing to this class and from there to the target class. (The intermediate coercions have to be valid.)
where	In a call to setIs, where to store the metadata defining the relationship. Default is the global environment for calls from the top level of the session or a source file evaluated there. When the call occurs in the top level of a file in the source of a package, the default will be the namespace or environment of the package. Other uses are tricky and not usually a good idea, unless you really know what you are doing.
classDef	Optional class definition for class , required internally when setIs is called during the initial definition of the class by a call to setClass. *Don't* use this argument, unless you really know why you're doing so.

Details

Arranging for a class to inherit from another class is a key tool in programming. In R, there are three basic techniques, the first two providing what is called "simple" inheritance, the preferred form:

1. By the contains= argument in a call to setClass. This is and should be the most common mechanism. It arranges that the new class contains all the structure of the existing class, and in particular all the slots with the same class specified. The resulting class extension is defined to be simple, with important implications for method definition (see the section on this topic below).
2. Making class1 a subclass of a virtual class either by a call to setClassUnion to make the subclass a member of a new class union, or by a call to setIs to add a class to an existing class union or as a new subclass of an existing virtual class. In either case, the implication should be that methods defined for the class union or other superclass all work correctly for the subclass. This may depend on some similarity in the structure of the subclasses or simply indicate that the superclass methods are defined in terms of generic functions that apply to all the subclasses. These relationships are also generally simple.
3. Supplying coerce and replace arguments to setAs. R allows arbitrary inheritance relationships, using the same mechanism for defining coerce methods by a call to setAs. The difference between the two is simply that setAs will require a call to as for a conversion to take place, whereas after the call to setIs, objects will be automatically converted to the superclass.

   The automatic feature is the dangerous part, mainly because it results in the subclass potentially inheriting methods that do not work. See the section on inheritance below. If the two classes involved do not actually inherit a large collection of methods, as in the first example below, the danger may be relatively slight.

   If the superclass inherits methods where the subclass has only a default or remotely inherited method, problems are more likely. In this case, a general recommendation is to use the setAs mechanism instead, unless there is a strong counter reason. Otherwise, be prepared to override some of the methods inherited.

With this caution given, the rest of this section describes what happens when coerce= and replace= arguments are supplied to setIs.

The coerce and replace arguments are functions that define how to coerce a class1 object to class2, and how to replace the part of the subclass object that corresponds to class2. The first of

these is a function of one argument which should be `from`, and the second of two arguments (`from`, `value`). For details, see the section on coerce functions below .

When by is specified, the coerce process first coerces to this class and then to `class2`. It's unlikely you would use the by argument directly, but it is used in defining cached information about classes.

The value returned (invisibly) by `setIs` is the revised class definition of `class1`.

Coerce, replace, and test functions

The `coerce` argument is a function that turns a `class1` object into a `class2` object. The `replace` argument is a function of two arguments that modifies a `class1` object (the first argument) to replace the part of it that corresponds to `class2` (supplied as `value`, the second argument). It then returns the modified object as the value of the call. In other words, it acts as a replacement method to implement the expression `as(object, class2) <- value`.

The easiest way to think of the `coerce` and `replace` functions is by thinking of the case that `class1` contains `class2` in the usual sense, by including the slots of the second class. (To repeat, in this situation you would not call `setIs`, but the analogy shows what happens when you do.)

The `coerce` function in this case would just make a `class2` object by extracting the corresponding slots from the `class1` object. The `replace` function would replace in the `class1` object the slots corresponding to `class2`, and return the modified object as its value.

For additional discussion of these functions, see the documentation of the `setAs` function. (Unfortunately, argument `def` to that function corresponds to argument `coerce` here.)

The inheritance relationship can also be conditional, if a function is supplied as the `test` argument. This should be a function of one argument that returns `TRUE` or `FALSE` according to whether the object supplied satisfies the relation `is(object, class2)`. Conditional relations between classes are discouraged in general because they require a per-object calculation to determine their validity. They cannot be applied as efficiently as ordinary relations and tend to make the code that uses them harder to interpret. *NOTE: conditional inheritance is not used to dispatch methods.* Methods for conditional superclasses will not be inherited. Instead, a method for the subclass should be defined that tests the conditional relationship.

Inherited methods

A method written for a particular signature (classes matched to one or more formal arguments to the function) naturally assumes that the objects corresponding to the arguments can be treated as coming from the corresponding classes. The objects will have all the slots and available methods for the classes.

The code that selects and dispatches the methods ensures that this assumption is correct. If the inheritance was "simple", that is, defined by one or more uses of the `contains=` argument in a call to `setClass`, no extra work is generally needed. Classes are inherited from the superclass, with the same definition.

When inheritance is defined by a general call to `setIs`, extra computations are required. This form of inheritance implies that the subclass does *not* just contain the slots of the superclass, but instead requires the explicit call to the coerce and/or replace method. To ensure correct computation, the inherited method is supplemented by calls to `as` before the body of the method is evaluated.

The calls to `as` generated in this case have the argument `strict = FALSE`, meaning that extra information can be left in the converted object, so long as it has all the appropriate slots. (It's this option that allows simple subclass objects to be used without any change.) When you are writing your coerce method, you may want to take advantage of that option.

Methods inherited through non-simple extensions can result in ambiguities or unexpected selections. If `class2` is a specialized class with just a few applicable methods, creating the inheritance

relation may have little effect on the behavior of class1. But if class2 is a class with many methods, you may find that you now inherit some undesirable methods for class1, in some cases, fail to inherit expected methods. In the second example below, the non-simple inheritance from class "factor" might be assumed to inherit S3 methods via that class. But the S3 class is ambiguous, and in fact is "character" rather than "factor".

For some generic functions, methods inherited by non-simple extensions are either known to be invalid or sufficiently likely to be so that the generic function has been defined to exclude such inheritance. For example initialize methods must return an object of the target class; this is straightforward if the extension is simple, because no change is made to the argument object, but is essentially impossible. For this reason, the generic function insists on only simple extensions for inheritance. See the simpleInheritanceOnly argument to setGeneric for the mechanism. You can use this mechanism when defining new generic functions.

If you get into problems with functions that do allow non-simple inheritance, there are two basic choices. Either back off from the setIs call and settle for explicit coercing defined by a call to setAs; or, define explicit methods involving class1 to override the bad inherited methods. The first choice is the safer, when there are serious problems.

References

Chambers, John M. (2016) *Extending R*, Chapman & Hall. (Chapters 9 and 10.)

Examples

```
## Two examples of setIs() with coerce= and replace= arguments
## The first one works fairly well, because neither class has many
## inherited methods do be disturbed by the new inheritance

## The second example does NOT work well, because the new superclass,
## "factor", causes methods to be inherited that should not be.

## First example:
## a class definition (see \link{setClass} for class "track")
setClass("trackCurve", contains = "track",
         slots = c( smooth = "numeric"))
## A class similar to "trackCurve", but with different structure
## allowing matrices for the "y" and "smooth" slots
setClass("trackMultiCurve",
         slots = c(x="numeric", y="matrix", smooth="matrix"),
         prototype = structure(list(), x=numeric(), y=matrix(0,0,0),

                               smooth= matrix(0,0,0)))
## Automatically convert an object from class "trackCurve" into
## "trackMultiCurve", by making the y, smooth slots into 1-column matrices
setIs("trackCurve",
      "trackMultiCurve",
      coerce = function(obj) {
        new("trackMultiCurve",
            x = obj@x,
            y = as.matrix(obj@y),
            smooth = as.matrix(obj@smooth))
      },
      replace = function(obj, value) {
        obj@y <- as.matrix(value@y)
        obj@x <- value@x
```

```
        obj@smooth <- as.matrix(value@smooth)
        obj})



## Second Example:
## A class that adds a slot to "character"
setClass("stringsDated", contains = "character",
         slots = c(stamp="POSIXt"))

## Convert automatically to a factor by explicit coerce
setIs("stringsDated", "factor",
      coerce = function(from) factor(from@.Data),
      replace= function(from, value) {
                  from@.Data <- as.character(value); from })

ll <- sample(letters, 10, replace = TRUE)
ld <- new("stringsDated", ll, stamp = Sys.time())

levels(as(ld, "factor"))
levels(ld) # will be NULL--see comment in section on inheritance above.

## In contrast, a class that simply extends "factor"
## has no such ambiguities
setClass("factorDated", contains = "factor",
         slots = c(stamp="POSIXt"))
fd <- new("factorDated", factor(ll), stamp = Sys.time())
identical(levels(fd), levels(as(fd, "factor")))
```

---

setLoadActions　　パッケージのロードに対するアクションを設定する

Description

これらの関数はパッケージの名前空間のロード中になされるべき計算を指定するパッケージ対する機構を提供する．そうしたアクションはロード時だけに利用可能な情報 (動的にリンクされるライブラリ中の位置のように) を提供する柔軟な方法である．

setLoadAction() または setLoadActions() の呼び出しは対応する名前空間がロードされる時に呼び出される一つまたはそれ以上の関数を指定し，...引数名はアクションに対する名前を指定するために使われる．

getLoadActions はパッケージの名前空間をその引数として与え，現在定義されているロードアクションを報告する．

hasLoadAction はもし与えられた名前に対応するロードアクションが where 名前空間に対して以前に設定されていれば TRUE を返す．

evalOnLoad() と evalqOnLoad() はロード時に指定された特定の表現式のスケジュールを立てる．

Usage

```
setLoadAction(action, aname=, where=)
```

```
setLoadActions(..., .where=)

getLoadActions(where=)

hasLoadAction(aname, where=)

evalOnLoad(expr, where=, aname=)

evalqOnLoad(expr, where=, aname=)
```

Arguments

action, ...	一つまたはそれ以上の引数の関数で，パッケージがロードされる時呼び出される．関数は一つの引数 (パッケージの名前空間)で呼び出されるので全ての引き続く引数は既定値を持たなければならない． もし ...の要素が名前付きならば，これらの名前は対応するロードメタデータに対して使われる．
where, .where	ロードアクションのリストがそれに対して定義されるパッケージの名前空間．この引数はもし呼び出しがパッケージ自体のソースコードに由来すれば普通省略されるが，パッケージが別のパッケージに対するロードアクションを提供すれば必要となる．
aname	アクションの名前．もしアクションが名前の提供なしに設定されると，既定では指定されたアクション系列中の位置が使われる (".1" 等)．
expr	環境 where 中のロードアクション中で評価される評価式．evalqOnLoad() の場合は評価式は文字通りに解釈される． evalOnLoad() 中のそれでは，典型的にはタイプ "language" のオブジェクトとして予め計算される必要がある．

Details

evalOnLoad() と evalqOnLoad() 関数は簡便性のために存在する．それらは表現式を評価し，そしてその関数への呼び出しをスケジュールするため setLoadAction() を呼び出す．

setLoadAction() または setLoadActions() への引数として提供された関数の各々は名前空間中のメタデータとして保存され，典型的には setLoadActions() の呼び出しを含むパッケージのそれになる．もしこのパッケージの名前空間がロードされると，これらの関数の各々が呼び出される．アクション関数はそれらが setLoadActions() に提供された順序で呼び出される．付値されたオブジェクトは呼び出し中で提供された名前から構成されたメタデータ名を持つ；名前無しの引数はそれらのアクションのリスト中の位置から命名される (".1" 等)．

setLoadAction() または setLoadActions() の複数の呼び出しをパッケージのコード中で使うことが出来る；アクションは setLoadAction() に与えられた名前が既存のアクションのそれである場合を除き，任意の先に指定された後にスケジュールされる．典型的なアプリケーションでは，パッケージ自体のコードから幾つかのアクションを設定するために呼び出される時 setLoadActions() がより便利である．もしアクション名が構成されるならば setLoadAction() がより便利であり，これは一つのパッケージが別のパッケージに対するロードアクションを構成するときにはより典型的である．

アクションは実際にか構成された同じ名前で逐次的に付値することで改訂できる．置き換えは依然適正な関数でなければならないが，勿論もし意図することが指定されたアクションを除くことにあれば何もしないことも可能である．

関数は少なくとも一つ引数を持たなければならない．それらはパッケージの名前空間という一つの引数で呼び出される．関数は S4 メタデータの処理の最後，任意のコンパイル済みコードの動的リンク，もしあれば .onLoad() の呼び出し，そしてメソッドのキャッシュとクラス定義の後に呼び出されるが，名前空間が封印される前である．(ロードアクションはメソッド選択適用がオンの時だけ呼び出される．)

関数は従って呼び出し中の引数として提供された名前空間中のオブジェクトに付値したり修正したり出来る．機構は，動的にリンクされたライブラリから得られる値のような，パッケージがロード時まで利用できない情報を保存することを可能にする．

ロードアクションは setHook() により提供されるユーザのロードフックと比較されるべきである．ユーザフックは一般にパッケージの外部から提供され，そして名前空間が封印された後に実行される．ロードアクションは普通パッケージコードの一部であり，アクションのリストは普通パッケージがインストールされる時に確立される．

ロードアクションはパッケージに対するソースコード中で直接に提供することが出来る．一つのパッケージ中に別のパッケージ中のロードアクションを作る機能を提供することも可能で有用である．ソフトウェアはアクション関数を正しい環境，つまり目標パッケージの名前空間，に付値されるように注意しなければならない．

Value

setLoadAction() と setLoadActions() はそれらの副作用のために呼び出され，そして有用な値は何も返さない．

getLoadActions() は提供された名前空間中のアクションの名前付きリストを返す．

hasLoadAction() はもし指定されたアクション名がこのパッケージに対するアクション中にあれば TRUE を返す．

See Also

ベースパッケージ中のより安全 (名前空間が封印された後に実行されるので)でより徹底的なバージョンに対しては setHook.

Examples

```
## Not run:
## あるパッケージに対するコード中で

## ... 何か他のもの
setLoadActions(function(ns)
   cat("Loaded package", sQuote(getNamespaceName(ns)),
       "at", format(Sys.time()), "\n"),
  setCount = function(ns) assign("myCount", 1, envir = ns),
  function(ns) assign("myPointer", getMyExternalPointer(), envir = ns))
  ... somewhere later
if(countShouldBe0)
  setLoadAction(function(ns) assign("myCount", 0, envir = ns), "setCount")

## End(Not run)
```

setMethod　　メソッドを作り保存する

Description

与えられた関数とクラスリストに対する形式的メソッドを作り保存する.

Usage

```
setMethod(f, signature=character(), definition,
          where = topenv(parent.frame()),
          valueClass = NULL, sealed = FALSE)

removeMethod(f, signature, where)
```

Arguments

f	総称的関数または関数の文字列名. 内部的に選択適用を行う非プリミティブな基本関数は文字列で指定される必要がある.
signature	対応するクラスの文字列名を持つ f に対する形式的引数名のマッチ. 下の詳細を見よ；もしシグネチャが自明でなければ, setMethod への適正な呼び出しを生成するためには method.skeleton を使うべきである.
definition	関数定義で, f への呼び出し中で引数が signature 中のクラスに直接または継承でマッチした時に呼び出されるメソッドになる.
where	メソッドの定義を保管する環境. setMethod に対しては, この引数を省略しそしてトップレベルで評価されるコードをソースコード中に含めることが推奨される；つまり, R セッション中で source に同値な何かか, パッケージに対する R ソースコードの一部として. removeMethod に対しては, 既定値はこのシグネチャに対するメソッドの(最初の)インスタンスの位置.
valueClass	旧式で使われないが, setGeneric に対する同じ引数を見よ.
sealed	もし TRUE ならば, そのように定義されたメソッドは setMethod への別の呼び出しで再定義されない (取り除いた後に再付値できるが).

Details

setMethod の呼び出しは, 典型的には大局的環境かパッケージの名前空間である環境中のこの総称的関数に対するメタデータテーブル中の提供されたメソッド定義を保管する. パッケージの場合, テーブルオブジェクトはパッケージの名前空間か環境の一部になる. パッケージが後のセッション中にロードされる時, メソッドは対応する総称的関数オブジェクト中のメソッドのテーブル中に混ぜ合わされる.

総称的関数は関数名とパッケージ名の組み合わせにより参照される；例えばパッケージ "methods" からの関数 "show". メソッドに対するメタデータは二つの文字列で特定される；特に総称的関数オブジェクト自体はその名前とそのパッケージ名を含むスロットを持つ. 総称的関数のパッケージ名はそれが元々そこから由来したパッケージに従って設定される；特に, そしてしばしば, 関数の非総称バージョンがそこから由来するパッケージ. 例えば, パッケージ **base** 中の全ての関数に対する総称的関数はパッケージ名として "base" を持つ. もっともそのパッケージについてはどれも S4 総称的関数ではないが.

これらは真の関数よりも，プリミティブである基本関数の殆どを含む；詳細については setGeneric に対するドキュメント中のプリミティブ関数に関する節を見よ．

複数のパッケージが同じ総称的関数に対してメソッドを持つことが出来る；つまり総称的関数名とパッケージ名の同じ組み合わせに対して．メソッドは別個の環境の別個のテーブル中に保管されるものの，対応するパッケージのロードはセッションの間中メソッドを総称的関数自体に加える．

シグネチャ中のクラス名は "numeric", "character" そして "matrix" の様な基本的クラスを含む任意の形式的クラスであり得る．二つの追加の特殊なクラス名を使うことが出来る；この引数が任意のクラスを持つことが出来ることを意味する "ANY"；そしてこの引数がこのシグネチャにマッチするために呼び出し中に現れてはならないことを意味する "missing" である．この二つを混同してはならない：もしある引数がシグネチャ中に述べられていないと，それは暗黙のうちにクラス "ANY" に対応し， "missing" に対してではない．下の例を見よ．旧式('S3')のクラスもまたこれらとの互換性を必要とすれば使うことが出来るが， S3 スタイルの継承が動作するためには setOldClass を呼び出し明確にこれらのクラスを宣言しなければならない．

メソッド定義は引数に対する既定表現式を持つことが出来るが，現在の限界は総称的関数は既定のメソッドが使われるためには同じ引数に対して ある既定表現式を持たねばならないということである．もしそうならば，そしてもし対応する引数が総称的関数の呼び出し中で欠損しているならば，メソッド中の既定表現式が使われる．もしメソッド定義が引数に対して如何なる既定値を持たなければ，総称的関数自体の定義中で提供される表現式が使われるが，この表現式は，総称的関数のそれでは無く，メソッドの囲み環境を用いて評価されることを注意する．またシグネチャ中のクラス "missing" の指定は如何なる既定表現式も必要とせず，そしてメソッド選択は既定の表現式を評価しないことを注意する．総称的関数のシグネチャ中の全ての実際の(欠損していない)引数は

メソッドが選択された時，standardGeneric(f) の呼び出しが起こった時，に評価されることを注意する．

setMethod に提供されたメソッドに対する形式的引数と総称的関数のそれらの間にある違いがあることが可能である．簡単に言えば，もし総称的関数が ... をその引数の一つとして持てば，メソッドは追加の引数を持つかもしれず，それは f への呼び出し中で ... にマッチする引数からマッチされる． (実際に起こることは修正された形式的引数をもつ局所的な関数がメソッド内部で作られ，そしてその局所的関数を呼び出すメソッドが再定義される．)

メソッドの選択適用は f に対して集められた利用可能なメソッドへの呼び出し中で実際の引数のクラスのマッチを試みる．もしこの呼び出し中に正確に同じクラスに対して定義されたメソッドがあれば，そのメソッドが使われる．さもなければ，実際のクラスか("ANY" を含む) 実際のクラスのスーパークラスに対応する全ての可能なシグネチャが考慮される．実際のクラスとの距離が最少のクラスが選択される；もし最少距離を持つクラスが複数あれば， (スーパークラスの言葉で辞書式順序で最初の)ものが選ばれるが警告が出る．全ての選択された継承メソッドは別のテーブルに保存されるので，継承計算は実際のクラス系列毎でセッション毎になされるだけで良い．より詳細は Methods を見よ．

関数 removeMethod は指定されたメソッドを対応する環境中のメタデータオブジェクトから取り除く．普通何の定義も残さないよりはメソッドを再定義したいため，これはそれほど使われる関数ではない．

Value

これらの関数は，指定された総称的関数に対するメソッドを定義するオブジェクト中のメソッドを設定したり取り除いたりする，それらの副作用のために存在する．

removeMethod が返す値はもし取り除かれるメソッドが見つかれば TRUE.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

`method.skeleton` は `setMethod` の呼び出しの骨格を作る推奨される方法であり，正しい形式的引数と他の詳細を備える．

一般的議論については Methods とそこでのリンク，“...” の関して選択適用を行うメソッドについては `dotsMethods`，そして総称的関数については `setGeneric`．

Examples

```
require(graphics)
## track オブジェクトのプロットに対するメソッド(setClass に対する例を見よ)
##
## 先ず引数として唯一つのオブジェクト：
setMethod("plot", signature(x="track", y="missing"),
  function(x,  y, ...) plot(slot(x, "x"), slot(x, "y"), ...)
)
## 次にtrackからのデータを y 軸上に任意の x データに対してプロット
setMethod("plot", signature(y = "track"),
 function(x, y, ...) plot(x, slot(y, "y"), ...)
)
## そして track を x 軸上に同様に
## (シグネチャに対する指定の短縮形を用いて)
setMethod("plot", "track",
 function(x, y, ...) plot(slot(x, "y"), y,  ...)
)
t1 <- new("track", x=1:20, y=(1:20)^2)
tc1 <- new("trackCurve", t1)
slot(tc1, "smooth") <- smooth.spline(slot(tc1, "x"), slot(tc1, "y"))$y
plot(t1)
plot(qnorm(ppoints(20)), t1)
## 継承メソッドと，それに準拠するメソッド引数
## (メソッド中の dotCurve 引数に注意，これは
## 総称的関数中の ... を取り出す)
setMethod("plot", c("trackCurve", "missing"),
function(x, y, dotCurve = FALSE, ...) {
  plot(as(x, "track"))
  if(length(slot(x, "smooth") > 0))
    lines(slot(x, "x"), slot(x, "smooth"),
         lty = if(dotCurve) 2 else 1)
  }
)
## tc1 だけのプロットは追加の曲線を持つ；tc1 の別の使用は
## それが "track" オブジェクトであるかのように扱われる．
plot(tc1, dotCurve = TRUE)
plot(qnorm(ppoints(20)), tc1)

## 特殊関数に対するメソッドの定義．
```

```
## "[" と "length" は普通の関数ではないが
## それらに対するメソッドを定義できる
setMethod("[", "track",
  function(x, i, j, ..., drop) {
    x@x <- x@x[i]; x@y <- x@y[i]
    x
  })
plot(t1[1:15])

setMethod("length", "track", function(x)length(x@y))
length(t1)

## メソッドは欠損引数に対しても同様に定義できる
setGeneric("summary") ## 関数を総称的関数にする

## summary() に対するメソッド
## メソッド定義は引数を含むことができるが,
## もしそれらが省略されると，クラス "missing" が仮定される.

setMethod("summary", "missing", function() "<No Object>")
```

---

setOldClass　　旧式スタイル*(S3)*のクラスと継承

Description

旧式('S3' としても知られる)のクラスを形式的に定義されたクラスとして登録する. Classes 引数は class 属性として使われる文字列ベクトル；特に，もし複数の文字列があれば，旧式のクラス継承が模倣される． setOldClass 経由の登録はメソッドシグネチャ中に S4 クラスのスロットか S4 クラスのスーパークラスとして S3 クラスが現れるのを可能にする.

Usage

```
setOldClass(Classes, prototype, where, test = FALSE, S4Class)
```

Arguments

Classes	S3 クラスの名前を与える文字列ベクトルで， S3 計算中の class 属性の付値の右辺に登場するようなもの. もし S3 クラスがそのデータがその形式であることが必要とされることが知られていれば， S3 クラスに加えてオブジェクトタイプまたは他の適正なデータ部分を指定できる.
prototype	プロトタイプとして使うオプションのオブジェクト．これはクラスに対する既定の S3 オブジェクトとして与えられるべきである．もし省略されると，S3 クラスを登録するために作られる S4 クラスは VIRTUAL である．詳細を見よ.
where	どこにクラス定義を保管するか，既定では大局的またはトップレベルの環境． (パッケージに対するソース中のどれかの関数が呼び出される時，クラス定義は既定でパッケージの環境中に含まれる．)

test
: フラグで，もし TRUE なら各オブジェクトに対する継承の明示的なテストを手配する，もし S3 クラスが同じ最初の文字列を持つ異なったクラス文字を持てるためには必要とされる．これは実装中の異なった機構でオプションの継承をもつクラスの各々の対に対して個別に指定されなければならない．

S4Class
: オプションの S4 クラスクラス定義かクラス名．新しいクラスはこのクラスの全てのスロットと他の性質，プラス Classses 引数で定義されるようなその S3 継承を持つ．引数 prototype と test はこの場合提供されてはならない．以下の“既知の属性を持つ S3 クラス”節を見よ．

Details

各々の名前は S4 クラスとして定義され， Classes 中の残りのクラスと全ての旧式のクラスの ‘ルート’を拡張する． S3 クラスは形式的定義を持たず，従って形式的に定義されたスロットは無い．もし setOldClass() の呼び出し中に prototype 引数が与えられると，クラスからのオブジェクトが new の呼び出しにより生成できる；しかしながらこれは普通サブクラスからオブジェクトを生成する程は妥当ではない (下の S3 クラスの拡張に関する節を見よ)．もしプロトタイプが提供されないと，クラスは仮想的 S4 クラスとして作られる．主要な欠点はこのクラスをスロットとして使う S4 クラス中のプロトタイプオブジェクトはそのスロット中に NULL オブジェクトを持ち，これはある場合に混乱に導く可能性がある．

バージョン 2.8.0 の R から (登録された) S3 クラスを新しい S4 クラスのスーパークラスに使うためのサポートが提供された．下の S3 クラスの節と例を見よ．

メソッドの選択適用と継承の詳細は Methods を見よ．

ある種の S3 クラスは S4 クラスとスーパークラスの通常の組み合わせとして表現できない，なぜなら S3 クラスからのオブジェクトはクラス中の文字列の可変なセットを持つことができるからである．そうしたクラスを S4 クラスとして登録することは依然として可能であるが，しかし継承は今や各オブジェクトに対して検証されなければならず，各スーパークラスに対して一度 setOldClass を引数 test=TRUE で呼び出さなければならない．

例えば，順序付き因子は 常に S3 クラス c("ordered", "factor") を持つ．これは適正な挙動であり， "factor" を "ordered" に拡張して単純に S4 クラスにマップする．

しかしそのクラス属性が "POSIXt" を最初の文字列としてもつオブジェクトは二番目の文字列として "POSIXct" または "POSIXlt" (またはどちらも) を持つかもしれない．この挙動は S4 クラス中にマップできるが，例えば is(x, "POSIXlt") を評価するために各オブジェクトに関して S3 クラス属性のチェックを必要とする．クラス定義中に明示的なテストが含められる．このテストを持つことは決してまずくはないが，しかしそれは継承クラスに対して定義されたメソッドに対して相当のオーバーヘッドを加えるので，もしオブジェクト固有のテストが必要であることが知られている時だけこの引数を与えるべきである．

リスト .OldClassesList は methods パッケージにより定義された旧式のクラスを含む．このリストの各要素は文字列ベクトルで，もし継承が含まれれば複数の文字列を持つ．リストの各要素は **methods** パッケージを作る時 setOldClass に渡された；従って，これらのクラスは呼び出し setMethod 中で使うことが出来，リストが含意する継承を持つ．

S3 クラスの拡張

setOldClass の呼び出しは S3 クラスに対応する形式的クラスを作り，他のクラスのスロットとしてや setMethod 中のシグネチャ中に許され， S3 継承を模倣する．

R 中の S4 クラスの最初の実装に対するドキュメントではユーザは S3 クラスを含む S4 クラスの定義に対しては，これらが登録されていたとしても，警告された．警告は主に2つ

の点に基づいていた． 1: オブジェクトの S3 挙動は S3 クラスが可視でないため失敗する，例えば S3 メソッドが選択適用された時． 2: S3 クラスは形式的定義を持たないので，そうしたクラスからのオブジェクトの S3 部分については一般に何も保証されない． (警告は下の最初の参照まで繰り返される．)

しかしながr，S3 クラスを含みそしてその挙動を拡張する S4 クラスを定義するのは多くの応用で魅力的である．形式的クラスとメソッド定義の柔軟さとセキュリティ無しに S3 プログラミングに忠実であることである．

バージョン 2.8.0 から，R は登録された S3 クラスの拡張に対するサポートを提供する；つまり， `contains=` 引数がその中に S3 クラスを含む `setClass` の呼び出しにより定義された新しいクラスに対して．サポートは第一にクラス oldClass を拡張する全てのクラスに対する S3 クラス情報を提供することを第一に目指している，特にそうしたクラスからの全てのオブジェクトが S3 クラスを特別なスロットに含むことを保証することで．

既存の S3 クラスへの拡張を指示する三つの異なった方法がある： `setOldClass()`, `setClass()` そして `setIs()` である．殆どの場合， `setOldClass` の呼び出しが最良のアプローチであるが，下で説明されたような特別な状況では別の方法が好まれるかもしれない．

`"A"` が `"oldClass"` を拡張する任意のクラスと仮定する．

```
setOldClass(c("B", "A"))
```

はその S3 クラスが `"B"` と `S3Class("A")` を連結する新しいクラス `"B"` を作る．新しいクラスは仮想的クラスである．もし `"A"` が既知の属性/スロットを使って定義されていないと， `"B"` はこれらのスロットも持つ；従ってクラス `"B"` からの対応する S3 オブジェクトが実際に要求される属性を真に持つことを確信しなければならない．追加の属性を指定するために新しいクラスに S4 定義を与えることが出来ることを注意する (次の節で解説されるように．) 最初の別の呼び出しは非仮想的なクラスを呼び出す．

```
setClass("B", contains = "A")
```

これはクラス `"A"` と同じスロットとスーパークラスを持つ非仮想的なクラスを作る．しかしながら，クラス `"B"` はプロトタイプ中に明示的にそれを与えない限り，新しいクラスの S3 クラススロットには含まれない．

```
setClass("B"); setIs("B", "A", .....)
```

これは `"A"` を拡張する仮想的なクラスを作るが， `"A"` のスロットを含まない． `setIs` への追加の引数は強制変換と置き換えメソッドを提供すべきである．新しいクラスが S3 メソッドを継承するためには，強制変換メソッドは作られたクラス `"A"` のオブジェクトが適当な S3 クラスを持つことを保証しなければならない．この三番目のアプローチを好む唯一のありそうな理由は `"B"` がクラス `"A"` 中の既知の属性と一貫性がないことである．

バージョン 2.9.0 の R 以来， S3 クラスを拡張するクラスからのオブジェクトはそのクラスに対して定義された S3 メソッドに渡される時対応する S3 クラスに変換される (つまり S3 クラス属性中の文字列の一つに対して)．これは通常の S3 オブジェクトに対して動作するならば，可能な限りそうしたメソッドが動作することを保証することを意図している．詳細は `Classes` を見よ．

既知の属性を持つ S3 クラス

S3 クラスの更なる特定を もしクラスが既知のクラスのある属性を持つことが保証されれば行うことが出来る (スロットに関しては， “既知”とは属性が指定されたクラスか，そのクラスのサブクラスであることを意味する).

この場合 `setOldClass()` の呼び出しは既知の構造を表現する S4 クラス定義を提供できる． S4 スロットは属性として実装されている (主として単にこの理由のため)ため，既知の属性は S4 クラスの表現中に指定できる．通常のテクニックは希望の構造を持つ S4 ク

ラスを作ることであり，それからクラス名か定義を setOldClass() への S4Class 引数として与えることである．

下の例のクラス "ts" の定義を見よ． S4 クラスを作る setClass の呼び出しは，そこでのように，クラス定義が封印されていない限り，同じクラス名を使うことが出来る．この例では，"ts" は "tsp" に対する数値スロットを持つベクトル構造として定義した．この定義の妥当性はこのクラスに対する全ての S3 コードが定義と一貫性があることに依存している；特に，全ての "ts" オブジェクトがベクトル構造として挙動し数値 "tsp" 属性を持つことである．我々は R 中の全ての基本コードに対してこれが正しいと信じているが， S3 クラスに対しては常にそうであるように保証は出来ない．

S4 クラス定義は S3 クラスがこれらと一貫性のある振る舞いをすることが保証されれば(例では，時系列オブジェクトは structure クラスと一貫性があることが保証される) 仮想的なスーパークラス("ts" のケース中のように) を持つことが出来る．

別の例では "data.frame" に対する S4 定義を探す．

S3 クラスがその保証された行動とそぐわぬ失敗は普通訂正されない． S3 クラスは本来定義を持たず，結果の不正な S4 オブジェクトはあらゆる面倒を引き起こしかねないからである．属性の存在やクラスのいずれかが保証されないため， (例えば配列中の dimnames，これらは S3 クラスでさえ無いが)，またはクラスは属性よりもリストの名前付き成分を使うため (例えば "lm")，多くの S3 クラスは既知のスロットの候補ではない．時として欠損する属性は，仮にそれがクラス "NULL" として存在するふりをしても，スロットとして表現できない．スロットでないような属性は値 NULL を持てないからである．
.

しかしながら，普通見逃される一つの例外はオプションで他の属性を存在が保証されている他の属性に付け加えることである (例えば model.frame が返す "data.frame" オブジェクト中の "terms")．バージョン 2.8.0 のように， validObject による正当性チェックは追加の属性を無視する；仮にこのチェックが将来厳格化されても，追加の属性は極めて普通なため，S3 クラスを拡張するクラスは恐らく免除されるであろう．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して：メソッド選択については節 10.6，そして総称的関数については節 13.4 を見よ).

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョン．)

See Also

setClass, setMethod

Examples

```
require(stats)
setOldClass(c("mlm", "lm"))
setGeneric("dfResidual", function(model)standardGeneric("dfResidual"))
setMethod("dfResidual", "lm", function(model)model$df.residual)

## dfResidual は lm オブジェクトと同様に mlm オブジェクトにも使える
myData <- data.frame(time = 1:10, y = (1:10)^.5)
myLm <- lm(cbind(y, y^3)  ~ time, myData)

showClass("data.frame")# 予め定義された S4 "oldClass" を見る

## S3 クラス "lm" を拡張する二つの例.
```

```
## クラス "xlm" は直接に，そして "ylm" は間接的に
setClass("xlm", representation(eps = "numeric"), contains = "lm")
setClass("ylm", representation(header = "character"), contains = "xlm")
ym1 = new("ylm", myLm, header = "Example", eps = 0.)
## より多くの例は ?\link{S3Class} を見よ.

utils::str(.OldClassesList)



## ベクトルを持つ S3 クラスが保証された属性 "stamped" と
## "date" 属性を持つ S3 クラスの例.
## 以下は生成関数と S3 プリントメソッド
## 注意：生成関数が属性クラスをチェックするのが本質的である
stamped <- function(x, date = Sys.time()) {
    if(!inherits(date, "POSIXt"))
      stop("bad date argument")
    if(!is.vector(x))
      stop("x must be a vector")
    attr(x, "date") <- date
    class(x) <- "stamped"
    x
}

print.stamped <- function(x, ...) {
    print(as.vector(x))
    cat("Date: ",  format(attr(x,"date")), "\n")
}

## そして同じ属性を持つ S4 クラス：
setClass("stamped4", contains = "vector", representation(date = "POSIXt"))

## "stamped" を登録するために S4 クラスをその属性付きで使うことが出来る：
setOldClass("stamped", S4Class = "stamped4")
selectMethod("show", "stamped")
## そしてそれから "stamped4" を取り除く
removeClass("stamped4")

someLetters <- stamped(sample(letters, 10),
                       ISOdatetime(2008, 10, 15, 12, 0, 0))

st <- new("stamped", someLetters)
st
# show() メソッドはオブジェクトのクラスをプリントし，
# それから S3 プリントメソッドを呼び出す.

stopifnot(identical(S3Part(st, TRUE), someLetters))

# S4 オブジェクトをそのデータ部分とスロットから直接に作る
new("stamped", 1:10, date = ISOdatetime(1976, 5, 5, 15, 10, 0))

## Not run:
## "ts" を S4 クラスとして定義する R 中のコード
setClass("ts", contains = "structure",
      representation(tsp = "numeric"),
      prototype(NA, tsp = rep(1,3)))
# プロトタイプは正規の S3 時系列
```

```
## そしてそれを S# クラスとして登録する
    setOldClass("ts", S4Class = "ts", where = envir)

## End(Not run)
```

---

show　　オブジェクトを示す*t*

Description

オブジェクトをプリント，プロットまたはそのクラスに適合する何ででも表示する．この関数はメソッドにより特定化されるために存在する．既定メソッドは showDefault を呼び出す．

show に対する形式的メソッドは普通自動プリント(詳細を見よ)のために起動される．

Usage

```
show(object)
```

Arguments

object	任意の R オブジェクト

Details

S4 クラス(setClass の呼び出しで定義されたクラス) のオブジェクトは show の呼び出しで自動的に表示される． S3 オブジェクトの属性として生じた S4 オブジェクトもこの形式で表示される；逆に， S4 オブジェクト中のスロットとして見つかった S3 オブジェクトは print の呼び出しによるかのように便宜的にプリントされる．

show に対して定義されたメソッドは単純な継承によってのみ継承される，さもなければメソッドは完全なオリジナルのオブジェクトを受け取り，誤解を招く結果を生じるからである．一般的な概念については setGeneric への simpleInheritanceOnly 引数と setIs 中の議論を見よ．

Value

show は不可視の NULL を返す．

See Also

showMethods は一つまたはそれ以上の関数に対する全てのメソッドをプリントする．

Examples

```
## setMethod のドキュメント中で紹介された例に続く ...
setClass("track",
         representation(x="numeric", y="numeric"))
setClass("trackCurve",
         representation("track", smooth = "numeric"))

t1 <- new("track", x=1:20, y=(1:20)^2)

tc1 <- new("trackCurve", t1)

setMethod("show", "track",
  function(object)print(rbind(x = object@x, y=object@y))
)
## メソッドは今や t1 の自動プリントに使われる

t1

## Not run:   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    2    3    4    5    6    7    8    9    10    11    12
y    1    4    9   16   25   36   49   64   81   100   121   144
  [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
x    13    14    15    16    17    18    19    20
y   169   196   225   256   289   324   361   400

## End(Not run)
## そして "track" を拡張するクラスのオブジェクトである tc1 に対しても
tc1

## Not run:   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    2    3    4    5    6    7    8    9    10    11    12
y    1    4    9   16   25   36   49   64   81   100   121   144
  [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
x    13    14    15    16    17    18    19    20
y   169   196   225   256   289   324   361   400

## End(Not run)
```

---

showMethods　　指定された関数またはクラス中に対する全てのメソッドを示す

Description

一つまたはそれ以上の総称的関数に対する要約を表示する，可能性として指定されたクラスを含むものに限られる.

Usage

```
showMethods(f = character(), where = topenv(parent.frame()),
            classes = NULL, includeDefs = FALSE,
            inherited = !includeDefs,
            showEmpty, printTo = stdout(), fdef)
.S4methods(generic.function, class)
```

Arguments

f	一つまたは複数の関数名．もし省略されると，他の引数にマッチする全ての関数が示される． 引数は単一の総称的関数に評価される表現式であり得，この場合引数 fdef は無視される．関数に対する表現式の提供は隠匿されていたり匿名の関数の吟味を許す； isDiagonal() に対する例を見よ．
where	もし引数として与えられないと，どこで総称的関数を探すか． f が欠損しているか長さが 0 なら，これは又どの総称的関数を調べるかを決定する．もし where が与えられると， getGenerics(where) により返される総称的関数だけがプリントに好ましい．もし where も欠損していれば，全てのキャッシュされた総称的関数が考慮される．
classes	もし引数 classes が提供されると，これは表示結果をそのシグネチャが一つまたは複数のこれらのクラスを含むようなメソッドに限定するクラス名のベクトル．
includeDefs	もし includeDefs が TRUE ならば，プリントアウト中に個々のメソッドの定義を含める．
inherited	このセッション中に関する限り，継承により見つかっていればメソッドを含め継承されているとマークされるかを指示する論理値．継承メソッドは普通それがこのセッション中で使われるまでは登場しない．どのメソッドが特定の引数のクラスに対して選択適用されるかを知りたければ selectMethod を見よ．
showEmpty	他の基準にマッチするメソッドが無いメソッドもともかく表示するかどうかを指示する論理値．既定では引数 f が欠損していない時そしてその時だけ TRUE．
printTo	それに対して情報が示されるコネクション；既定では標準出力に出力される．
fdef	オプションで，使用する総称的関数の定義；もし欠損していれば，指定されていれば where の中を探す． ‘詳細’節中のコメントも見よ．
generic.function, class	methods を見よ．

Details

.S4methods の記述に対しては methods を見よ．

総称的関数の名前とパッケージは，様々な引数により決定される基準に従って，メソッドがそれに対して現在定義されているシグネチャのリストが続く．パッケージは総称的関数のソースを参照することを注意する．総称的関数に対する個々のメソッドは同様に他のパッケージに由来できる．

指定されるか f が欠損しているため，複数の総称的関数が含まれるときは，関数が探され，引数 fdef としての総称的関数を含み， showMethods が各々に対して再呼び出しされる．複雑な状況では，これはある種の変則的な結果を避けることが出来る．

Value

もし printTo が FALSE ならば，プリントされるべき文字列が返される；さもなければ値は invisible を使ったコネクションかファイル名．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

setMethod，メソッドを含む他のツールに対しては GenericFunctions； selectMethod は特定の関数と引数に対するクラスのシグネチャに対して選択適用されるメソッドを示す．

methods は軽量の対話的使用に対するメソッド発見ツールを提供する．

Examples

```
require(graphics)

## help(setMethod) の例の中でのように plot に対するメソッドを仮定する，
## クラス "track" を含むメソッドを(定義無し)プリント：
showMethods("plot", classes = "track")
## Not run:
# 関数 "plot":
# x = ANY, y = track
# x = track, y = missing
# x = track, y = ANY

require("Matrix")
showMethods("%*%")# many!
    methods(class = "Matrix")# 何もない
showMethods(class = "Matrix")# 全て
showMethods(Matrix:::isDiagonal) # 移出されていない総称的関数

## End(Not run)



if(no4 <- is.na(match("stats4", loadedNamespaces())))
   loadNamespace("stats4")
showMethods(classes = "mle") # -> show() に対するメソッド
if(no4) unloadNamespace("stats4")
```

---

signature-class　　メソッド定義に対するクラス *list("signature")*

Description

このクラスは関数の形式的引数のどれかから対応するクラスへのマッピングを表す．これは MethodDefinition クラス中の二つのスロットに対して使われる．

クラスからのオブジェクト

オブジェクトは形式 new("signature", functionDef, ...) の呼び出しにより作ることが出来る．もし関数オブジェクトとして与えられると， functionDef 引数は形式的な名前を定義する．他の引数はクラスを定義する．より典型的には，オブジェクトはメソッド定義の副作用として作られる．どちらの方法でも，普通対応するクラス定義が存在するので，クラスは適格に定義されていることが期待される． package スロットに関するコメントを見よ．

スロット

.Data: クラス名のクラス "character" のオブジェクト．

names: 対応する引数名のクラス "character" のオブジェクト．

package: クラス "character" のオブジェクトでクラス名に対応するパッケージ名．クラス名とパッケージの組み合わせはクラスをユニークに決定する．原則として，同じクラス名が複数のパッケージ中に現れるかもしれない，この場合シグネチャが上手く定義されるためには package 情報が必要とされる．

拡張

データ部分から，クラス "character"．クラス "character" から，クラス "vector"．

メソッド

initialize signature(object = "signature")：このクラスからのオブジェクトに対する議論は上を見よ．

See Also

このクラスの用法についてはクラス MethodDefinition.

---

slot 形式的クラスからのオブジェクト中のスロット

Description

これらの関数はオブジェクト中の個別のスロットに関する情報を返すか設定する．

Usage

```
object@name
object@name <- value

slot(object, name)
slot(object, name, check = TRUE) <- value
.hasSlot(object, name)

slotNames(x)
getSlots(x)
```

Arguments

object	形式的に定義されたクラスからのオブジェクト.
name	スロット名. 演算子は固定された名前を取り，それは言語中の構文的な名前ならば引用化されていなくて良い. スロット名は任意の空でない文字列で良いが，もし名前が文字，数，そして . から成り立っていなければ，引用化 (バックチックまたは一重・二重引用符で)される必要がある. slot 関数の場合，name はクラス定義中の任意の適正なスロットを評価する表現式で良い. 一般的に，単純な演算子ではなくこの関数を使う唯一の理由はスロット名を計算しなければならない からである.
value	名前付きスロットに対する新しい値. 値はこのオブジェクトのクラスに対して適正でなければならない.
check	slot の置き換えバージョンではフラグ. もし TRUE なら，このスロットの値としての付値される値の適正さをチェックする. 結果のオブジェクトが不正になり得るため，ユーザのコードはこれを FALSE に設定すべきではない.
x	クラスの名前 (文字列として)かクラス定義. もし文字列でもなくクラス定義でもない引数が与えられると， slotNames (だけ)は代わりに class(x) を使う.

Details

クラス定義はそのクラスに対して定義されたスロットを直接的または間接的に指定する. 各スロットは名前と関連するクラスを持つ. スロットの取り出しはそのクラスからのオブジェクトを返す. スロットの設定は先ず値を指定されたスロットに強制変換しそれからそれを保存する.

一般的な属性とは異なり，スロットは特にマッチされず，スロットにそのクラスに対しては不正な名前を要求(または設定を試みると) するとエラーになる.

@ 抽出演算子と slot 関数自体はクラス定義に対してチェックをせず，単にオブジェクト自体中の名前をマッチする. 置き換え形式はチェックを行う (check=FALSE の場合の slot を除く). スロットがインチキ無しに設定されている限り，取り出されたスロットは適正である.

インチキをする二つの方法があることを注意する. しかしどちらも避けるべきであり保証はされない. 自明な方法はスロットを check=FALSE で付値する. 又 R 中のスロットは，ある種の後方互換性のために，属性として実装されている. 現在の実装は属性を attr<- を使って付値されることを妨げず，そうした付値はスロット名の適正さをチェックされない.

取り出しと置き換えに対する "@" 演算子はプリミティブで実際は **base** パッケージ中に収まっている.

"@" と slot() の置き換えバージョンは付値の右辺側をスロットの宣言されたクラスへの変換の計算で異なる. どちらも提供された値が宣言されたスロットクラスのサブクラスからのものかどうかを検証する. slot() バージョンはもしあれば強制変換メソッドの呼び出しに移り，実際には計算 as(value, slotClass, strict = FALSE) を行う. "@" バージョンは関係を検証し，任意の強制変換の実行を後に残す (例えば，関連するメソッドが選択適用される時).

殆どの使用で結果は同値であり， "@" バージョンは追加の関数呼びしを保存するが，経験的な証拠が変換が必要なことを示すと，置き換え前に as() を呼び出すか slot() の置き換えバージョンを使う.

Value

"@" 演算子と slot 関数はオブジェクトに対する形式的に定義されたスロットから取り出したり，置き換えたりする．

関数 slotNames と getSlots はそれぞれ指定されたクラス定義中のスロットの名前とスロットに関連したクラスを返す．その x (上)の拡張された解釈を除けば， slotNames(x) は正に names(getSlots(x)) である．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

See Also

@, Classes, Methods, getClass, names.

Examples

```
setClass("track", representation(x="numeric", y="numeric"))
myTrack <- new("track", x = -4:4, y = exp(-4:4))
slot(myTrack, "x")
slot(myTrack, "y") <- log(slot(myTrack, "y"))
utils::str(myTrack)

getSlots("track") # または
getSlots(getClass("track"))
slotNames(class(myTrack)) # 次と同じ
slotNames(myTrack)
```

StructureClasses 基本構造に対応するクラス

Description

仮想クラス structure とそれを拡張するクラスは配列と時系列の様な S3 言語構造の形式的クラス類似物である．

Usage

```
## 次のクラス名はメソッドのシグネチャ中に
## as() や is() 表現中のクラスと，VIRTUAL とコメントされた
## クラスを除き，new() への呼び出し中に登場できる

"matrix"
"array"
```

"ts"

"structure" ## VIRTUAL

クラスからのオブジェクト

オブジェクトは new(Class, ...) の形式の呼び出しで作ることが出来る，ここで Class は特定のクラスの引用符付きの名前で (例えば "matrix")，他の引数はもしあれば対応する関数への引数と解釈される，例えば関数 matrix() に対するもの．多重のクラスに対して動作するようにデザインされた恐らくクラス名と渡される引数付きのソフトウェアを書くのでなければ，これらの関数を直接呼び出すことに対する特別な利点はない．

クラス "matrix" と "array" から作られたオブジェクトは控え目に言っても普通でない状態が続いている．それらはこれらのクラスからのオブジェクトに見えるが， S3 や S4 クラスオブジェクトとしての内部構造を持たない．特にそれらは "class" 属性を持たずクラスを持つオブジェクトとは認識されない (つまり is.object と isS4 は共にそうしたオブジェクトに対して FALSE を返す)．しかしながらこれらの擬似クラスに対するメソッド(S4 と S3 双方)が定義でき，新しいクラス(S4 と S3 双方)はそれらを継承できる．

オブジェクトがあたかも対応するクラス由来であるかのように依然として振る舞う (とにかく多くの場合)ことは R の基本コード中に組み込まれたそうしたオブジェクトを認識する特別なコードに由来する．多くの目的に対してこれらのクラスを普通のように扱うことは幸運にも動作する．特殊な扱いの一つの結果がこれらの二つのクラスを S4 クラスのデータ部分に使うことが出来ることである；例えば "matrix" のサブクラスである S4 クラスを作る setGeneric への呼び出し中で contains = "matrix" を用い避けることが出来ることである．全てが完全に動作する保証はないが，幾つかのクラスがこの形式で成功裡に書かれている．

"matrix" や "array" を含むクラスはそのクラスを持つ .Data スロットを持つことを注意する．これはオブジェクトのタイプを指示する擬似クラスとして以外の .Data の唯一の用法である．この場合オブジェクトのタイプは含まれる行列か配列のタイプになる．一般的な議論は Classes を見よ．

クラス "ts" は setOldClass 機構を用いて S4 として登録されている本質的に S3 クラスである．バージョン 2.7.0 の R はこれを純粋の S4 クラスとして扱っていた．これは原則的に良いアイデアであったが，実際にはサブクラスの定義を許さず他の本質的な問題を抱えていた．(例えば， "tsp" パラメータをスロットとして設定することは，組み込みの実装はスロットがデータの長さに関して一時的に非一貫的であることを認めないためにしばしば失敗する．また S4 クラスはクラス "mts" に対する S3 継承の正しい指定を妨げた．)

時系列オブジェクトは，行列や配列とは対照的に， S4 スタイルの定義を使い正当な S3 クラス "ts" を持つ (これがどのようになされるかは setOldClass のドキュメントの例を見よ)．パッケージ **stats** 中の "mts" の S3 継承は同様に登録されている．これらのクラスは， "matrix" と "array" と同様に，新しい S4 クラスに対するスーパークラスとして殆どの例で適正であるべきである．

これらのクラスの全ては可能な限り基本生成関数 matrix, array そして ts と同じ引数を受け入れる initialize に対する特別な S4 メソッドを持つ．制限は一つより多くの非仮想的なスーパークラスを持つクラスは new への呼び出し中でそのスーパークラスからのオブジェクトを受け入れなければならないということである；従ってそうしたクラス (ある種の言語では “mixin” と呼ばれる)は特別な引数を持たない initialize に対する既定メソッドを使う．

拡張

個別のクラスは全て "structure" により直接にクラス "structure" そしてクラス "vector"

を拡張する．

メソッド

coerce 任意のオブジェクトをこれらのクラスに強制変換するメソッドを対応する基本関数を呼び出すことにより定義できる．例えば as(x, "matrix") は as.matrix(x) を呼び出す．もし as() の呼び出しで strict = TRUE ならば，メソッドは全ての他のスロットと dim と dimnames 以外の属性を取り除こうとする．

Ops グループメソッド(例えば S4groupGeneric を見よ) は構造とベクトル (特殊例として配列と行列)に対して定義され，参考文献中のようにベクトル構造を実装する．本質的に，ベクトルと結びついた構造は結果のオブジェクトが同じ長さを持つ限り構造を保つ．他の構造と結び付けられた構造は，構造を定義するスロットに何が起きるべきか決定する自動的な方法は無いため，構造を取り除く．

これらのメソッドは構造クラスのどれかかクラス "vector" を継承するクラスを含むパッケージがロードされると活性化される．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して．)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して．)

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (オリジナルのベクトル構造に対して).

See Also

別のモデルを強要するクラス nonStructure，そこではもし任意の数学変換または操作が基本クラスのどれかを拡張するクラスに適用されると全てのスロットが取り除かれる．

Examples

```
showClass("structure")

## 少し調べてみる：
showClass("ts")
(ts0 <- new("ts"))
str(ts0)

showMethods("Ops") # これらのクラスからの6種類のメソッド，もしかすると更に多く
```

---

testInheritedMethods　継承メソッドの選択をテスト，報告する

Description

異なった継承シグネチャのセットが指定された総称的関数の全てのメソッドに対する継承をテストするために生成される．もしメソッド選択がこれらのどれかに対して曖昧ならば，返されるオブジェクトに曖昧さの要約が付け加えられる．このテストはパッケージをリリースする前にパッケージ作者により実行されるべきである．

Usage

```
testInheritedMethods(f, signatures, test = TRUE, virtual = FALSE,
                     groupMethods = TRUE, where = .GlobalEnv)
```

Arguments

f	総称的関数またはそれの文字列名．既定ではこの総称的関数に対する全てのメソッドシグネチャの現在定義されている全てのサブクラスが調べられる．他の引数は主にどのような継承パターンが調べられるかを修正するオプションである．
signatures	testInheritedMethods により計算される関連サブクラスの代わりに使うサブクラスシグネチャのオプションのセット．これがどのようになされるかについては詳細を見よ．この引数は選択をバッチでテストするために test = FALSE を使った呼び出しの後に提供されるかもしれない．
test	メソッド選択を実際にテストするかどうかを制御するオプションのフラグ．もし FALSE ならば各シグネチャに対して selectMethod を呼び出すこと無しにサブクラスに対する関連シグネチャのリストを単に返す．もし非常に沢山のシグネチャがあれば，完全なリストを集めそれからバッチでそれらをテストしたくなるかもしれない．
virtual	関連サブクラスの中に仮想的クラスを含めるべきか．実際の引数のクラスだけが総称的関数の呼び出し中で継承の計算を惹き起こすので，普通はしない．仮想的クラスを含めることは，もしクラスが現在非仮想的なサブクラスを持たないがユーザが将来そうしたクラスを定義するかもしれないと予想するのならば，有用かもしれない．
groupMethods	グループ総称的関数に対するメソッドも含めるべきか?
where	クラス定義をその中で探す環境．ほとんど常に，関連するメソッドと/またはクラス定義をもつ全てのパッケージを付加した後に既定の大局的環境を使おう．

Details

以下の解説は普通のケースであるオプション引数が省略された場合に適用される．先ず全てのメソッドに対するシグネチャの定義は findMethodSignatures の呼び出しで計算される．これらから全ての既知の非仮想的サブクラスがあるメソッドのシグネチャ中に現れる各クラスに対して見つけられる．これらのサブクラスはそれらがどのクラスを継承するかに従ってグループに分割され，そして各グループからただ一つのサブクラスが保持される (総称的関数のシグネチャ中の各引数にたいして)．従ってもしメソッドがある引数に対するクラス "vector" を用いて定義されていると，一つの実際のベクトルクラスが任意に選択される． "ANY" のケースは全てのクラスがそれを拡張するため特別に扱われる．非仮想的クラスであるダミーの ".Other" はテストされるもののうちスーパークラスを持たない全てのクラスに対応する．

総称的関数のシグネチャ中の引数に対して保持されたサブクラスの全ての組み合わせがそれから計算される．結果の行列の各行は selectMethod の呼び出しによりテストされるシグネチャである．曖昧な選択に関する情報を収集するため， testInheritedMethods は特別なシグナル "ambiguousMethodSelection" に対する呼び出しハンドラーを，対応するオプションを設定することで，確立する．

Value

クラス "methodSelectionReport" のオブジェクト．このクラスの詳細は現在変更の可能性がある．これはスロット "target", "selected", "candidates" そして "note" を持ち，全ては曖昧なケースを参照する (そしてもし何もなければ長さは 0 である)．これらのスロットはプログラマにより検出とできれば曖昧なメソッド選択をフィックスすることが意図されている．オブジェクトは更に総称的関数名であるスロットと，そしてテストされる全てに対するラベルのベクトルを与える "allSelections" を含む．

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (メソッド選択の基本については節10.6.)

Chambers, John M. (2009) *Class Inheritance in R* https://statweb.stanford.edu/~jmc4/classInheritance.pdf.

Examples

```
## もし他の付加されたパッケージが
## `+` またはそのグループ総称的関数に対するメソッドを持たなければ,
## これは選択パターンの 16 x 2 行列を返す(R 2.9.0 では)
testInheritedMethods("+")
```

TraceClasses　　トーレスを制御するために内部的に使われるクラス

Description

ここで解説されるクラスは R 関数 trace によりブラウザー呼び出しを含む関数とメソッドのバージョンを作るのと，そしてまた同じオブジェクトを untrace するのに使われる．

Usage

```
### 以下のクラスからのオブジェクトは名前に "WithTrace" を
### 持たない対応するオブジェクトに trace() を呼び出すことで作られる．

"functionWithTrace"
"MethodDefinitionWithTrace"
"MethodWithNextWithTrace"
"genericFunctionWithTrace"
"groupGenericFunctionWithTrace"

### 次は上のクラスの各々により拡張される仮想的クラス

"traceable"
```

クラスからのオブジェクト

オブジェクトがこれらのクラスから trace への呼び出しで作られる．(クラス "traceable" に対する initialize メソッドがあるがそれを直接使う必要はありそうもない．)

スロット

.Data: データ部分で，クラス "functionWithTrace"，そして他のクラスに対する "function".

original: オリジナルのクラスのオブジェクト；つまりクラス "functionWithTrace" に対する "function".

拡張

クラスの各々は対応する未トレースクラスをデータ部分から拡張する；例えば "functionWithTrace" は "function" を拡張する．特殊なクラスの各々は "traceable" を直接に拡張し，そしてクラス "VIRTUAL" は "traceable" により拡張される．

メソッド

特殊なクラスのポイントは関数 trace() によりそれらから生成されたオブジェクトはそれらの新しいトレース情報に加えて呼び出し可能または選択不可能に留まることである．

See Also

関数 trace

---

validObject　　オブジェクトの適正さをテストする

Description

object のそのクラス定義に関する適正さがテストされる．もしオブジェクトが適正なら TRUE が返される；さもなければ不適切さを説明する文字列ベクトルが返されるか，エラーが生じる(test が TRUE かどうかに従って)．オプションで，オブジェクト中の全てのスロットがまた検証される．

関数 setValidity はクラスの検証メソッドを設定する (しかしより普通には，このメソッドは setClass の validity 引数として提供される)．メソッドは TRUE か非適切の記述を返す一つのオブジェクトの関数であるべきである．

Usage

```
validObject(object, test = FALSE, complete = FALSE)

setValidity(Class, method, where = topenv(parent.frame()) )

getValidity(ClassDef)
```

Arguments

object	任意のオブジェクトだが，オブジェクトが形式的定義を持たない限りあまり何も起きない．
test	論理値；もし TRUE で不適切とされれば，この関数は問題を説明する文字列ベクトルを返す．もし test がFALSE (既定値)ならば正当性検証の失敗はエラーを惹き起こす．

complete	論理値；もし TRUE ならば検証メソッドはそうしたメソッドを持つスロット全てに再帰的に適用される.
Class	検証メソッドが設定されるクラスの名前かクラス定義.
ClassDef	クラス定義オブジェクトがで， getClassDef に由来するようなもの.
method	検証メソッド；つまり，NULL か引数(object)が一つの関数. validObject と同様に，関数はもしオブジェクトが適正であれば TRUE を，もし問題があれば一つまたはそれ以上の説明文字列を返すべきである. validObject とは異なり，これはエラーを発生すべきではない.
where	修正されたクラス定義はこの環境中に保管される. 検証メソッドはスーパークラスの適正さをチェックする必要はないことを注意する：validObject のロジックはこれらのテストが唯一回行われることを保証する. 結論として，もし一つの検証メソッドが別のを使いたければ，それは getValidity() を呼び出して他のクラスの他の定義からメソッドを取り出して呼び出すべきである：それは validObject を呼び出すべきでは無い.

Details

検証テストは‘ボトムアップ’で行われる：オプションで，もし complete=TRUE ならば，もしあればオブジェクトのスロットの適切さがテストされる. それから，全てのケースで，このクラスが拡張するクラス (‘スーパークラス’)の各々に対して，もしあればそのクラスの明示的な検証メソッドが呼び出される. 最後に，もしあれば，object のクラスの検証メソッドが呼び出される.

あるスロットがその適正さのテストで失敗しても全てのスロットが検査されるものの，テストは一般にエラーを見つけた最初の段階で停止する.

標準の検証テスト(complete=FALSE を用いる)がオブジェクトが任意の追加引数で new により作られた時適用される (余分の引数無しでは結果は単にクラスのプロトタイプオブジェクトである).

もしその引数が object でなければ検証メソッドの定義をフィックスする試みがなされる.

Value

validObject はもしオブジェクトが適切ならば TRUE を返す. さもなければ見つかった問題を説明する文字列ベクトルを返すが，もし test が FALSE ならば適正さの失敗は対応するエラーメッセージ文字列を伴うエラーを引き起こす.

References

Chambers, John M. (2008) *Software for Data Analysis: Programming with R* Springer. (R バージョンに対して.)

Chambers, John M. (1998) *Programming with Data* Springer (オリジナルの S4 バージョンに対して.)

See Also

setClass; class classRepresentation.

Examples

```
setClass("track",
         representation(x="numeric", y = "numeric"))
t1 <- new("track", x=1:10, y=sort(stats::rnorm(10)))
## 適正な "track" オブジェクトは同じ数の x, y 値を持つ
validTrackObject <- function(object) {
    if(length(object@x) == length(object@y)) TRUE
    else paste("Unequal x,y lengths: ", length(object@x), ", ",
               length(object@y), sep="")
}
## 関数をクラスに対する適正さメソッドとして付値する
setValidity("track", validTrackObject)
## t1 は適正な "track" オブジェクトであるべきである
validObject(t1)
## ここで何かまずいことを行う
t2 <- t1
t2@x <- 1:20
## これはエラーを起こすべき
## Not run: try(validObject(t2))


setClass("trackCurve",
         representation("track", smooth = "numeric"))

## validObject が引数付きの initialize() から呼び出される時
## 全てのスーパークラス適正メソッドが使われるので，これは失敗する
## Not run: trynew("trackCurve", t2)


setClass("twoTrack", representation(tr1 = "track", tr2 ="track"))

## 適正さのテストは既定では再帰的に適用されないので，
## このオブジェクトが作られる(不適切に)
tT  <- new("twoTrack", tr2 = t2)

## より厳格なテストは問題を検出する
## Not run: try(validObject(tT, complete = TRUE))
```

Index