A practical guide to BitBake

Harald Achitz

## 1. Prefazione

### 1.1. Il tutorial

Chi sta leggendo questo tutorial probabilmente già sa che BitBake è usato come strumento per il building, principalmente da OpenEmbedded e dallo Yocto Project, per le distribuzioni Linux. È noto anche che lavorare con BitBake ha una curva di apprendimento piuttosto ripida. Questo documento ha lo scopo di appiattirla un po'.

Non è una guida completa per BitBake, non è possibile in così poco spazio, ma si cerca di spiegarne alcune delle funzioni fondamentali. Queste basi saranno d'aiuto quando si cominceranno a scrivere le ricette di BitBake.

### 1.2. Obbiettivi

Il tutorial mostra come creare il progetto più piccolo possibile per poi estenderlo, passo dopo passo, per illustrare come funziona BitBake

### 1.3. Riconoscimenti

Grazie alla [Tritech](http://tritech.se/) per avermi concesso del tempo per preparare le basi per questo documento nel 2014. Da allora, questo tutorial si è evoluto, ma senza la primissima versione questo documento non esisterebbe.

Molte grazie alle persone che segnalano problemi ed errori di battitura all'["issue tracker" di questo sito](https://bitbucket.org/a4z/bitbakeguide/issues)!

### 1.4. Feedback

Se si trovano bug, sezioni poco chiare, errori o semplicemente si hanno dei suggerimenti, si prega di usare "the issue tracker" su:  
<https://bitbucket.org/a4z/bitbakeguide/issues>  
Non è richiesta alcuna registrazione.

Si può inoltre commentare con Disqus alla fine del documento.

## 2. BitBake

### 2.1. Cos'è BitBake

Per lavorare con BitBake aiuta sapere questo:  
Fondamentalmente BitBake è un programma Python che, guidato dalla apposita configurazione, esegue task [attività] per specifici target, le cosiddette ricette.

#### 2.1.1. Config, task e ricette

Configurazioni, task e ricette sono scritte in una specie di DSL (Domain Specific Language) di BitBake contenenti variabili e codice eseguibile Shell o Python. In teoria, dal momento che BitBake esegue del codice, qualcuno potrebbe pensare di usarlo per qualcos'altro, oltre alla creazione di software, ma non è una buona idea.

BitBake è stato creato come tool per il build del software ed ha quindi delle funzionalità speciali, ad esempio la possibilità di definire delle dipendenze. BitBake è in grado di risolvere le dipendenze ed eseguire le cose nell'ordine giusto. Inoltre, il build di pacchetti software richiede spesso di eseguire le stesse attività o certe altre molto simili. I task [attività] comuni sono ad esempio: scaricare ed estrarre il codice sorgente, eseguire 'configure', eseguire 'make' o semplicemente scrivere un messaggio nel log. BitBake ha un meccanismo per astrarre, incapsulare e riutilizzare tali funzionalità in modo configurabile.

## 3. Il setup di BitBake

Bitbake è scaricabile da:  
<https://github.com/openembedded/bitbake>  
Si seleziona un branch e una versione e si scarica lo zip. Lo si estrae in qualche cartella e, da quella stessa posizione, verrà creata la cartella bitbake-$version.

|  |  |
| --- | --- |
|  | Questo tutorial è stato aggiornato e testato per funzionare con **python 3.8.8** e **bitbake 1.52.0**. [Tutti gli esempi](https://bitbucket.org/a4z/bitbakeguide) sono stati adattati per funzionare con tali versioni, allo stato attuale, all'inizio del 2022. |

Questo tutorial è stato inizialmente scritto diversi anni fa. Da allora, sia Python che BitBake hanno apportato modifiche incompatibili e ciò potrebbe accadere di nuovo in futuro. Eventuali problemi vanno segnalati nella sezione commenti o tramite l'[issue tracker](https://bitbucket.org/a4z/bitbakeguide/issues). In caso di problemi, per continuare a lavorare con questo tutorial, utilizzare BitBake e Python nelle versioni menzionate.

|  |  |
| --- | --- |
|  | Se utilizzato all'interno di Yocto non è necessario installare BitBake, in quanto è già incluso con suoi sorgenti. Yocto richiede la creazione di uno script che faccia ciò che si andrà a fare ora, installando BitBake nel proprio ambiente. |

### 3.1. Installazione di BitBake

L'installazione è semplicissima:

* Si aggiunge la directory bitbake-$version/bin al PATH
* Si aggiunge la directory bitbake-$version/lib a PYTHONPATH

Lo si fa eseguendo

export PATH=/path/to/bbtutor/bitbake/bin:$PATH  
export PYTHONPATH=/path/to/bbtutor/bitbake/lib:$PYTHONPATH

Questo è fondamentalmente lo stesso dello script init di Yocto.  
Tale script crea anche una cartella build, qui lo si farà in seguito.

Prima si controlla che tutto funzioni e che BitBake sia installato.  
Per farlo si esegue il seguente comando di BitBake:

bitbake --version

Stamperà qualcosa come:

BitBake Build Tool Core version 1.52.0

### 3.2. La documentazione di BitBake

Oltre a quella online, la documentazione viene fornita anche con il codice sorgente.

In un terminale ci si posiziona nella cartella bitake-$version/doc e si esegue

make html DOC=bitbake-user-manual

per creare il file ...doc/\_build/html/bitbake-user-manual/bitbake-user-manual.html.

La documentazione ufficiale dell'utente può essere letta parallelamente a questo tutorial, ad esempio, per cercare informazioni aggiuntive, se necessario. In ogni caso, si dovrebbero leggere i documenti ufficiali dopo aver letto questo tutorial.

La [Documentazione dello Yocto Project](https://docs.yoctoproject.org/bitbake/singleindex.html) ha una sezione per la versione di BitBake utilizzata.

## 4. Creazione di un progetto

### 4.1. Layout del progetto BitBake

Solitamente un progetto BitBake è organizzato in cartelle con la configurazione e i metadati, chiamate layer, e una cartella di build.

#### 4.1.1. La cartella del layer

Una cartella layer contiene le descrizioni della configurazione, dei task e del target per quello che fa BitBake.  
È pratica comune chiamare tale cartella meta-'qualcosa'.

#### 4.1.2. La cartella di build

La cartella di build è quella in cui viene eseguito il comando bitbake. Qui, BitBake si aspetta il file di configurazione iniziale e vi collocherà tutti i file che crea.

Per eseguire BitBake senza errori bisogna creare una cartella per i layer e inserirvi dei file configurazione.

### 4.2. Il progetto minimo

La configurazione minima sarà simile a questa:

bbTutorial/  
├── build  
│   ├── bitbake.lock  
│   └── conf  
│   └── bblayers.conf  
└── meta-tutorial  
 ├── classes  
 │   └── base.bbclass  
 └── conf  
 ├── bitbake.conf  
 └── layer.conf

La directory di build sarà la nostra directory di lavoro, può avere un nome qualsiasi, ma è una buona pratica anteporre al nome "build".

Il nome meta-tutorial segue la convenzione dei nomi dei layer. Vedremo meglio in seguito.

Questi 4 file

* bblayers.conf
* base.bbclass
* bitbake.conf
* layer.conf

verranno creati successivamente.

#### 4.2.1. I file config necessari

Prima una descrizione dei file necessari, poi quella del loro contenuto.

build/conf/bblayers.conf

Il primo file che BitBake si aspetta è conf/bblayers.conf nella sua directory di lavoro, che è la nostra directory build.  
Per il momento lo si crea con questo contenuto:

build/conf/bblayers.conf

BBPATH := "${TOPDIR}"  
BBFILES ?= ""  
BBLAYERS = "${TOPDIR}/../meta-tutorial"

meta-tutorial/conf/layer.conf

Ogni layer deve avere un file conf/layer.conf. Per il momento lo si crea con questo contenuto:

meta-tutorial/conf/layer.conf

BBPATH .= ":${LAYERDIR}"  
BBFILES += "${LAYERDIR}/recipes-\*/\*/\*.bb"

meta-tutorial/classes/base.bbclass  
meta-tutorial/conf/bitbake.conf

Per ora, questi file possono essere presi dalla directory di installazione di BitBake.  
Questi file si trovano nelle cartelle *bitbake-$version/conf* e *bitbake$version/classes*.  
Basta semplicemente copiarli nel progetto del tutorial.

|  |  |
| --- | --- |
|  | Usando BitBake dalla distribuzione Poky, questi file non ci sono. In tal caso, prelevarli [da qui](https://github.com/openembedded/bitbake) |

#### 4.2.2. Qualche nota sui file creati

build/conf/bblayers.conf

* Aggiungere la directory di lavoro corrente a BBPATH assegnandola a TOPDIR.  
  TOPDIR è internamente impostata da BitBake con la directory di lavoro corrente.
* Inizializzare la variabile BBFILES come vuota. Le ricette verranno aggiunte in seguito.
* Aggiungere il path di meta-tutorial alla variabile BBLAYERS.  
  Una volta in esecuzione, BitBake cercherà ulteriori configurazioni in tutte le directory dei layer noti.

meta-tutorial/conf/layer.conf

* LAYERDIR è una variabile che BitBake passa al layer che sta caricando.  
  Questo path va accodato alla variabile BBPATH.
* BBFILES indica a BitBake dove sono le ricette.  
  Per ora non aggiungiamo nulla, perché non abbiamo ancora ricette, vedremo in seguito.

|  |  |
| --- | --- |
|  | ".=" e "+=" accodano il valore ad una variabile senza o con uno spazio. Questa, e altre notazioni, sono documentate nel capitolo 3.1..\* della documentazione di BitBake |

conf/bitbake.conf

conf/bitbake.conf contiene un mucchio di variabili che, per ora, si prendono così come sono.

classes/base.bbclass

Un file \*.bbclass contiene delle funzionalità shared [condivise/in comune].  
Questo base.bbclass contiene alcune funzioni per il log, che si userà in seguito, e un task per il build che non fa nulla.  
Non è molto utile, ma è richiesto da BitBake poiché build è l'attività che esegue per default se non ne viene specificata una. In seguito si cambierà questa funzione.

|  |  |
| --- | --- |
|  | La sezione 3.3 del manuale di BitBake descrive le "Sharing Functionality" [Funzionalità Condivise]. |

#### 4.2.3. Il path di ricerca di BitBake

Per BitBake ci sono alcuni path che sono relativi a BBPATH.  
Ciò significa che se si chiede a BitBake di cercare un path, lo farà in tutte le directory in BBPATH. (Da notare, come PATH, BBPATH possono contenere più directory, separate da ':'. )  
Avendo aggiunto TOPDIR e LAYERDIR a BBPATH, classes/base.bbclass e conf/bitbake.conf potrebbe trovarsi in ognuno di essi.  
Ma ovviamente sono stati aggiunti alla directory del meta-tutorial.  
La directory di build non dovrebbe mai contenere file generici, solo quelli speciali come local.conf che è valido solo per l'effettiva build. In seguito useremo un local.conf.

### 4.3. La prima esecuzione

In un terminale, ci si sposta nella cartella *build* appena creata, che è la directory di lavoro.  
Si eseguirà sempre bitbake dalla cartelle di build in modo che bitbake possa trovare il relativo file *conf/bblayers.conf*.

Ora è sufficiente eseguire bitbake senza alcun argomento.

[~/bbTutorial/build]

bitbake

Se le impostazioni sono corrette bitbake riporterà

Nothing to do. Use 'bitbake world' to build everything, or run 'bitbake --help' or usage information.

Non è molto utile, ma è un punto di partenza.

Questa è una buona opportunità per introdurre un utile flag al comando: verbose, per emettere alcuni output di debug.

Per vederlo in azione si esegue

[~/bbTutorial/build]

bitbake -vDDD world

L'output sarà simile a questo:

NOTE: Cache: default: Not using a cache. Set CACHE = <directory> to enable.  
Loading cache: 100%  
Loaded 0 entries from dependency cache.  
DEBUG: collating packages for "world"  
DEBUG: Target list: []  
NOTE: Resolving any missing task queue dependencies  
DEBUG: Resolved 0 extra dependencies

|  |  |
| --- | --- |
|  | L'argomento ` -vDDD` dice a BitBake di essere il più prolisso possibile, e world è il target per il build. In seguito vedremo altre informazioni sui target. |

Dato che c'è una ricetta e un world [mondo] come target, bitbake esce con un errore. Per ora va bene, verrà sistemato nel prossimo capitolo.

Si è notato che BitBake crea una directory tmp?

|  |  |
| --- | --- |
|  | Tutto il codice di esempio è disponibile su <https://bitbucket.org/a4z/bitbakeguide> L'esempio per questo capitolo è ch04. |

## 5. La prima ricetta

BitBake ha bisogno di ricette per fare qualcosa. Al momento non ce ne sono, quindi anche se si esegue il comando bitbake, senza una ricetta non è molto divertente.

Si può facilmente verificare che non c'è nulla da fare eseguendo

[~/bbTutorial/build]

bitbake -s

Questo comando riporterà:

NOTE: Not using a cache. Set CACHE = <directory> to enable.  
Recipe Name Latest Version Preferred Version  
=========== ============== =================

Questo indica 2 cose:

1. BitBake dice che non ha una cache definita.
2. BitBake, mostrandoci una lista vuota, ci dice che non ha proprio nulla da fare

### 5.1. La posizione della cache

BitBake memorizza le informazioni temporanee dei metadati in una directory, la cache. Con questa si velocizzano le successive esecuzione dei comandi.

Si può riparare alla mancanza della cache aggiungendo semplicemente una variabile a bitbake.conf.  
Pertanto si modifica il file *meta-tutorial/conf/bitbake.conf* e si aggiunge alla fine:

meta-tutorial/conf/bitbake.conf

...  
CACHE = "${TMPDIR}/cache/default"

Per il momento è sufficiente.

|  |  |
| --- | --- |
|  | Nei progetti reali, come Yocto, questa variabile è già impostata e non ce ne si deve preoccupare. Spesso il path della cache è composto da diverse variabili per avere nel nome la configurazione della build effettiva (debug o release). |

Il passo successivo consiste nell'aggiungere una ricetta. Ciò richiede 2 passaggi:

1. consentire a BitBake di trovare le ricette
2. scrivere una prima ricetta

### 5.2. Aggiungere il path di una ricetta al layer del tutorial

BitBake ha bisogno di sapere quali ricette contiene un layer.  
Si modifica il file *meta-tutorial/conf/layer.conf* e si dice a BitBake di caricare tutti i file delle ricetta usando un pattern comune.

meta-tutorial/conf/layer.conf

BBPATH .= ":${LAYERDIR}"  
BBFILES += "${LAYERDIR}/recipes-\*/\*/\*.bb"

Si utilizza ora la variabile precedentemente definita in *build/conf/bblayers.conf*. Un file di ricetta ha l'estensione \*.bb e, se si rispetta il pattern comune, si possono aggiungere tutte le ricette a BitBake con una solo riga.

Solitamente le ricette sono raggruppate in una propria cartella, il che significa mettere le ricette che sono in qualche modo correlate nella stessa directory.

|  |  |
| --- | --- |
|  | Di norma queste cartelle si chiamano recipes-'gruppo', dove il nome del gruppo è quello di una categoria di programmi. |

Ora, dato che BitBake sa dove cercare le ricette, se ne può effettivamente aggiungere una.

Seguendo il pattern comune si creano le cartelle *meta-tutorial/recipes-tutorial/first* e qui si crea la prima ricetta. Anche le ricette hanno uno schema comune di nomenclatura che è {ricetta}\_{versione}.bb.

### 5.3. Creare la **prima** ricetta e il primo task

La prima ricetta stamperà semplicemente un messaggio nel log. La si mette nel primo gruppo chiamandola 'first', versione 0.1.

Quindi la ricetta **first** nel primo gruppo è:

meta-tutorial/recipes-tutorial/first/first\_0.1.bb

DESCRIPTION = "I am the first recipe"  
PR = "r1"  
do\_build () {  
 echo "first: some shell script running as build"  
}

* Il **task** do\_build sovrascrive quello globale della build, base.bbclass, che è vuoto.
* **PR** è il numero interno della revisione che si dovrebbe aggiornare dopo ogni cambiamento.
* La **description** dovrebbe essere autoesplicativa.

Se tutto è fatto correttamente si può chiedere a BitBake di elencare le ricette disponibili.

[~/bbTutorial/build]

bitbake -s  
Parsing recipes: 100% ...  
Parsing of 1 .bb files complete...  
Recipe Name Latest Version Preferred Version  
=========== ============== =================  
first :0.1-r1

e lo si può lanciare, sempre dalla directory di build

[~/bbTutorial/build]

bitbake first

Ora, controllando *tmp/work/first-0.1-r1/temp*, si troveranno molti file interessanti, ad esempio:

build/tmp/work/first-0.1-r1/temp/log.do\_build

DEBUG: Executing shell function do\_build  
first: some shell script running as build  
DEBUG: Shell function do\_build finished

|  |  |
| --- | --- |
|  | Tutto il codice di esempio è disponibile su <https://bitbucket.org/a4z/bitbakeguide> L'esempio per questo capitolo è ch05. |

## 6. Classi e funzioni

I prossimi passi saranno:

* Aggiungere una classe
* Aggiungere una ricetta che usi tale classe.
* Esplorare le funzioni

### 6.1. Creazione della classe mybuild

Si sta per creare e 'sharare' [pubblicare] una funzione diversa di build.  
Lo si può fare creando una classe nel layer del tutorial.  
Poi creando un nuovo file chiamato *meta-tutorial/classes/mybuild.bbclass*.

meta-tutorial/classes/mybuild.bbclass

addtask build  
mybuild\_do\_build () {  
  
 echo "running mybuild\_do\_build."  
  
}  
  
EXPORT\_FUNCTIONS do\_build

Come in base.class, si aggiunge un task per il build. Ancora una volta è una semplice funzione shell.  
Il prefisso *mybuild\_do\_* è per seguire le convenzioni, *nomeclasse\_do\_nomefunzione* per un task in una classe.

Per l'istruzione addtask BitBake aggiunge il prefisso *do\_* al nome della funzione, se non c'è già. Ecco perché, nell'esempio sopra, viene utilizzato addtask build. Si può scrivere anche addtask do\_build, per BitBake è lo stesso.

EXPORT\_FUNCTIONS rende disponibile la funzione di build agli utenti di tale classe.  
Se non ci fosse questa riga, non sarebbe possibile sovrascrivere la funzione di build in un secondo momento.

Per il momento c'è tutto per utilizzare questa classe con la seconda ricetta.

### 6.2. Utilizzo di myclass con la ricetta **second**

È ora di creare una ricetta "second" che utilizzerà il task definito in mybuild.  
Si suppone che questo target debba eseguire una patch prima del build.  
E come ulteriore sfida, "second" utilizzerà un po' anche Python.

Seguendo la nomenclatura di BitBake, si aggiunge una nuova cartella di ricette, inserendovi il file *meta-tutorial/recipes-tutorial/second/second\_1.0.bb*.

Il nuovo file sarà simile a questo.

meta-tutorial/recipes-tutorial/second/second\_1.0.bb

DESCRIPTION = "I am the second recipe"  
PR = "r1" (1)  
inherit mybuild (2)  
  
def pyfunc(o): (3)  
 print(dir(o))  
  
python do\_mypatch () { (4)  
 bb.note ("runnin mypatch")  
 pyfunc(d) (5)  
}  
  
addtask mypatch before do\_build (6)

|  |  |
| --- | --- |
| **1** | DESCRIPTION e PR sono stati descritti. |
| **2** | La classe mybuild viene ereditata e quindi myclass\_do\_build ne diventa il task di build di default. |
| **3** | La funzione (in puro Python) pyfunc accetta degli argomenti ed esegue la funzione dir di Python con tali argomenti e stampa il risultato. |
| **4** | La funzione mypatch (in Python di BitBake) viene aggiunta e registrata come task da eseguire prima della funzione di build. |
| **5** | mypatch chiama pyfunc e passa la variabile bitbake globale d. d (datastore) è definita da BitBake ed è sempre disponibile. |
| **6** | La funzione mypatch è registrata come task da eseguirsi prima della funzione di build. |

Ora c'è un esempio di utilizzo delle funzioni Python.

|  |  |
| --- | --- |
|  | La parte del manuale sulle funzioni di BitBake è la sezione 3.4. |

### 6.3. Esplorare ricette e task

Avendo ora due ricette, si è in grado di utilizzare ed esplorare ulteriori opzioni del comando bitbake.

Si possono ottenere informazioni sulle ricette e i loro task e controllare cosa eseguirà BitBake.

#### 6.3.1. Lista delle ricette e dei task

Per prima cosa si può verificare che BitBake abbia davvero entrambe le ricette. Lo si può fare con l'opzione *-s*.

[~/bbTutorial/build]

bitbake -s

verrà prodotto (alla fine)

...  
  
Recipe Name Latest Version Preferred Version  
=========== ============== =================  
first :0.1-r1  
second :1.0-r1

|  |  |
| --- | --- |
|  | Ci sarà un WARNING sulla compatibilità dei layer. Per il momento lo si ignora. Se ne parlerà nel prossimo capitolo. |

Volendo vedere tutti i task di una ricetta, possiamo farlo con bitbake -c listtasks second

Questo dovrebbe fornire una prima panoramica su come esplorare ricette e task.

### 6.4. Esecuzione dei task e build del mondo

Ci sono diverse opzioni per eseguire le build o degli specifici task per le ricette.

La build di una ricetta

Per eseguire tutti i task della seconda ricetta, si chiama semplicemente bitbake second

Esecuzione di un task

Si può anche eseguire un singolo task di una ricetta.  
Si supponga di voler eseguire solo l'attività mypatch della ricetta second.  
Lo si può fare col comando bitbake -c mypatch second

La build di tutto

Con bitbake world si eseguono tutte le attività per tutte le ricette

Si possono modificare i comandi e vedere cosa succede.  
L'output della console dirà cosa è stato fatto.

#### 6.4.1. Verifica nei log della build

Bitbake crea una directory tmp/work, nella sua posizione di build effettiva, in cui archivia tutti i file di log.  
Tali log contengono informazioni interessanti e vale la pena studiarli.

Un output dopo una prima esecuzione di bitbake world potrebbe essere questo.

tmp/work/  
 |- first-0.1-r1  
 |- temp  
 |-log.do\_build -> log.do\_build.20703  
 |-log.do\_build.20703  
 |-log.task\_order  
 |-run.do\_build -> run.do\_build.20703  
 |-run.do\_build.20703  
 |- second-1.0-r1  
 |- second-1.0  
 |- temp  
 |-log.do\_build -> log.do\_build.20706  
 |-log.do\_build.20706  
 |-log.do\_mypatch -> log.do\_mypatch.20705  
 |-log.do\_mypatch.20705  
 |-log.task\_order  
 |-run.do\_build -> run.do\_build.20706  
 |-run.do\_build.20706  
 |-run.do\_mypatch -> run.do\_mypatch.20705  
 |-run.do\_mypatch.20705

Questi file di log contengono informazioni utili su BitBake, sulle sue azioni e sull'output dei task eseguiti.

|  |  |
| --- | --- |
|  | Tutto il codice di esempio è disponibile su <https://bitbucket.org/a4z/bitbakeguide> L'esempio per questo capitolo è ch06. |

## 7. I layer di BitBake

Un tipico progetto BitBake è costituito da più layer.

Di solito i layer contengono ricette per un argomento specifico. Ad esempio il sistema di base, quello grafico, ecc.

In alcuni progetti potrebbero esserci anche più target per le build e ciascuno composto da layer diversi.  
Un tipico esempio potrebbe essere la creazione di una distribuzione Linux con e senza le componenti della GUI.

I layer si possono utilizzare, estendere, configurare ed è anche possibile sovrascrivere parzialmente parti di quelli esistenti.  
Questo consente il riutilizzo e la personalizzazione secondo le effettive esigenze.

Lavorare con più layer è il caso comune e quindi si va ad aggiungere anche un altro layer al progetto.

### 7.1. Aggiungere un ulteriore layer

L'aggiunta di un nuovo layer può essere fatto seguendo i seguenti passi:

1. Creare la cartella per il nuovo layer
2. Creare la configurazione del layer
3. Informare BitBake del nuovo layer
4. Aggiungere le ricette al layer

#### 7.1.1. Per aggiungere la cartella del nuovo layer

Si crea una nuova cartella chiamata *meta-two*.  
Seguendo le comuni convenzioni di nomenclatura, la struttura ora si presenta così:

ls ~/bbTutorial  
build meta-tutorial meta-two

#### 7.1.2. Configurazione del nuovo layer

Si aggiunge il file *meta-two/conf/layer.conf*. Questo file è esattamente come quello nel layer del tutorial.

meta-two/conf/layer.conf

BBPATH .= ":${LAYERDIR}"  
BBFILES += "${LAYERDIR}/recipes-\*/\*/\*.bb"

#### 7.1.3. Informare BitBake delle ricette del nuovo layer

editare build/conf/bblayers.conf ed estendere la variabile BBLAYERS.

build/conf/bblayers.conf

BBLAYERS = " \  
 ${TOPDIR}/../meta-tutorial \  
 ${TOPDIR}/../meta-two \  
"

### 7.2. l comando bitbake-layers

Per esplorare le configurazioni dei layer c'è il comando bitbake-layers.

bitbake-layers ha diverse opzioni, l'help dice tutto.

Available commands:  
 help  
 display general help or help on a specified command  
 show-recipes  
 list available recipes, showing the layer they are provided by  
 show-cross-depends  
 figure out the dependency between recipes that crosses a layer boundary.  
 show-appends  
 list bbappend files and recipe files they apply to  
 flatten  
 flattens layer configuration into a separate output directory.  
 show-layers  
 show current configured layers  
 show-overlayed  
 list overlayed recipes (where the same recipe exists in another layer)

Per vedere, ad esempio, quali layer esistono, si può eseguire

[~/bbTutorial/build]

bitbake-layers show-layers

Ma prima che questo produca qualsiasi output utile per il progetto, di devono adeguare le configurazioni dei layer.

### 7.3. Estendere la configurazione del layer

Si devono aggiungere delle informazioni alla configurazione del layer.

* un nome per l'elenco dei layer
* un pattern di ricerca per i file da aggiungere
* una priorità per il layer

Si inizia con *meta-tutorial/conf/layer.conf* e si aggiunge

meta-tutorial/conf/layer.conf

# append layer name to list of configured layers  
BBFILE\_COLLECTIONS += "tutorial"  
# and use name as suffix for other properties  
BBFILE\_PATTERN\_tutorial = "^${LAYERDIR}/"  
BBFILE\_PRIORITY\_tutorial = "5"

Le variabili utilizzate sono ben descritte nel manuale utente di BitBake, quindi non è necessario ripeterle.

I pattern dovrebbero essere chiari, si definisce il nome del layer e lo si usa come suffisso per altre variabili.  
Questo meccanismo, che utilizza suffissi del dominio definiti dall'utente nei nomi delle variabili, viene utilizzato da BitBake in diversi posti.

Si modificano, quindi, meta-two/conf/layer.conf nello stesso modo.

meta-two/conf/layer.conf

BBFILE\_COLLECTIONS += "two"  
BBFILE\_PATTERN\_two = "^${LAYERDIR}/"  
BBFILE\_PRIORITY\_two = "5"  
LAYERVERSION\_two = "1"

Se ora eseguendo bitbake-layers show-layers si ottiene

layer path priority  
========================================================  
meta-tutorial /path/to/work/build/../meta-tutorial 5  
meta-two /path/to/work/build/../meta-two 5

|  |  |
| --- | --- |
|  | Tutto il codice di esempio è disponibile su <https://bitbucket.org/a4z/bitbakeguide> L'esempio per questo capitolo è ch07. |

### 7.4. Compatibilità del layer

Un progetto come Yocto è composto da moltissimi layer. Per garantire che i layer utilizzati siano compatibili con una data versione del progetto, si può definire un nome, un *"seriale" del layer* e i layer possono indicare di essere compatibili con una o più *"seriali" di layer*.

In pratica, per lo Yocto Project, ogni release definisce un nome della release: il *layer series core name* [nome centrale della serie di layer]. I layer testati per la versione data possono aggiungere il "nome di compatibilità" nel proprio config. Se viene aggiunto un layer che non ha il "nome di compatibilità" specificato, bitbake mostrerà un warning.

Lo si può facilmente verificare. Finora non è stato specificato il "layer series core name" nel tutorial. Eseguendo, per esempio, bitbake-layers show-recipes si otterranno 5 warning.

bitbake-layers show-recipes  
NOTE: Starting bitbake server...  
WARNING: Layer tutorial should set LAYERSERIES\_COMPAT\_tutorial in its conf/layer.conf file to list the core layer names it is compatible with.  
WARNING: Layer two should set LAYERSERIES\_COMPAT\_two in its conf/layer.conf file to list the core layer names it is compatible with.  
WARNING: Layer tutorial should set LAYERSERIES\_COMPAT\_tutorial in its conf/layer.conf file to list the core layer names it is compatible with.  
WARNING: Layer two should set LAYERSERIES\_COMPAT\_two in its conf/layer.conf file to list the core layer names it is compatible with.  
Parsing recipes: 100% |##############################################################################################################################| Time: 0:00:00  
Parsing of 2 .bb files complete (0 cached, 2 parsed). 2 targets, 0 skipped, 0 masked, 0 errors.  
WARNING: No bb files matched BBFILE\_PATTERN\_two '^/home/bitbakeguide/ch07/build/../meta-two/'  
  
Summary: There were 5 WARNING messages shown.  
=== Available recipes: ===  
first:  
 meta-tutorial 0.1  
second:  
 meta-tutorial 1.0

I primi 4 warning si riferiscono al fatto che il progetto non ha i "seriali" di compatibilità dei layer. Il quinto warning si ha perché il layer "two" è vuoto, cosa che verrà risolta nel prossimo capitolo. Innanzitutto, si aggiunge un nome come "seriale", specificando che i 2 layer nel tutorial sono compatibili.

#### 7.4.1. "Core Name" della serie di layer

Prima di tutto si aggiunge un "Core Name" del progetto. Lo si fa impostando la variabile LAYERSERIES\_CORENAMES. In Yocto, viene fatto in un layer chiamato *core*.  
Si definisce il nome nel layer del tutorial perché è il primo layer. L'effettiva posizione non ha importanza, lo si potrebbe anche definire nel file build/conf/bblayers.conf.

Il "core name" si imposta aggiungendo

LAYERSERIES\_CORENAMES = "bitbakeguilde"

al file ...meta-tutorial/conf/layer.conf

#### 7.4.2. Il "seriale" per la compatibilità dei layer

Si deve anche specificare che il layer del tutorial è compatibile con la bitbakeguilde.  
Lo si può fare settando la variabile LAYERSERIES\_COMPAT\_…​ nel file layer.conf di ciascun layer. La variabile finisce sul nome del layer come si è visto con con la variabile BBFILE\_PATTERN or the BBFILE\_PRIORITY.

I file di configurazione dei layer ora sono così:

meta-tutorial/conf/layer.conf

BBPATH .= ":${LAYERDIR}"  
BBFILES += "${LAYERDIR}/recipes-\*/\*/\*.bb"  
BBFILE\_COLLECTIONS += "tutorial"  
BBFILE\_PATTERN\_tutorial = "^${LAYERDIR}/"  
BBFILE\_PRIORITY\_tutorial = "5"  
  
LAYERSERIES\_CORENAMES = "bitbakeguilde"  
  
LAYERVERSION\_tutorial = "1"  
LAYERSERIES\_COMPAT\_tutorial = "bitbakeguilde"

meta-two/conf/layer.conf

BBPATH .= ":${LAYERDIR}"  
BBFILES += "${LAYERDIR}/recipes-\*/\*/\*.bb \  
${LAYERDIR}/recipes-\*/\*/\*.bbappend"  
  
BBFILE\_COLLECTIONS += "two"  
BBFILE\_PATTERN\_two = "^${LAYERDIR}/"  
BBFILE\_PRIORITY\_two = "5"  
LAYERVERSION\_two = "1"  
LAYERDEPENDS\_two = "tutorial"  
  
LAYERSERIES\_COMPAT\_two = "bitbakeguilde"

Con queste modifiche i quattro warning sulle informazioni mancanti della compatibilità scompaiono. Tutti i layer vengono dichiarati compatibili con le serie dei "core layer".

### 7.5. Dipendenze dei layer

Si potrebbe aver notato che è stata valorizzata anche la variabile LAYERDEPENDS\_two nel file layer.conf del secondo layer, meta-two.

In questo modo si informa BitBake che tale layer ha una dipendenza da quello del tutorial. Si vedrà nel prossimo capitolo, quando si aggiungeranno più contenuti al layer meta-two, perché è stato fatto così.

|  |  |
| --- | --- |
|  | Provare a vedere cosa succede se non si imposta un "compat name", o lo si scrive male, e poi si esegue il comando show-layers con i diversi argomenti. |

## 8. Condividere e riutilizzare le configurazioni

Finora sono state utilizzate classi e file di configurazione per incapsulare la configurazione e i task.  
Ma ci sono altri modi per riutilizzare ed estendere i task e le configurazioni.  
Questi sono:

* ereditarietà delle classi
* I file bbappend
* I file "include"

Per mostrarne gli usi si aggiungerà un'ulteriore classe al layer-two.  
La nuova classe introdurrà una catena configure-build e riutilizzerà la classe mybuild esistente sfruttandone l'ereditarietà.  
Poi si userà questa nuova classe in una nuova ricetta.  
Successivamente si amplierà una ricetta esistente utilizzando la tecnica "append".

### 8.1. Ereditarietà della classe

Per realizzare la catena [sequenza] di configurazione/build si crea una classe che eredita dalla mybuild aggiungendo semplicemente un task di configurazione come dipendenza dal task della build.

Lo si crea come classe ulteriore, che si userà poi per mostrare una classe e una ricetta che fanno uso dell'ereditarietà.

meta-two/classes/confbuild.bbclass

inherit mybuild (1)  
  
confbuild\_do\_configure () { (2)  
  
 echo "running configbuild\_do\_configure."  
  
}  
  
addtask do\_configure before do\_build (3)  
  
EXPORT\_FUNCTIONS do\_configure (4)

|  |  |
| --- | --- |
| **1** | usa la classe mybuild come base |
| **2** | crea una nuova funzione |
| **3** | definisce l'ordine delle funzioni, configura prima della build |
| **4** | esegue l' "export" della funzione in modo che diventi disponibile |

Ora lo si può semplicemente usare nella ricetta **third** ed usare confbuild.

meta-two/recipes-base/third/third\_0.1.2.bb

DESCRIPTION = "I am the third recipe"  
PR = "r1"  
inherit confbuild

Questa ricetta eredita la classe confbuild.

Se ora si lancia bitbake third verranno eseguiti i task di configurazione e di build per terzo.

### 8.2. I file bbappend

Un file "append" si usa per aggiungere delle funzioni ad una ricetta esistente. L'aggiunta avviene tramite la corrispondenza del nome del file della ricetta. Il contenuto del file "append" viene accodato ad una ricetta con lo stesso nome.

Per poter utilizzare i file "append" si deve impostare il layer affinché li carichi assieme alle normali ricette.  
Poi si cambia la configurazione del layer e si aggiunge il caricamento del file \*.bbappend alla variabile BBFILES.

meta-two/conf/layer.conf

BBFILES += "${LAYERDIR}/recipes-\*/\*/\*.bb \  
${LAYERDIR}/recipes-\*/\*/\*.bbappend"

Ora si riutilizza ed si estende la prima ricetta esistente.

|  |  |
| --- | --- |
|  | Questo è il motivo per cui è stato aggiunto il LAYERDEPENDS\_two nel capitolo precedente, ce n'è bisogno perché contiene la ricetta da estendere. |

Quello che si vuole, è eseguire una patch prima del build, quindi si deve creare il file bbappend corrispondente e aggiungere le modifiche.  
Poi c'è da creare la cartella *meta-two/recipes-base/first/* e il file *first\_0.1.bbappend*.

meta-two/recipes-base/first/first\_0.1.bbappend

python do\_patch () {  
 bb.note ("first:do\_patch")  
}  
  
addtask patch before do\_build

Se ora si elencano i task della prima ricetta, si vedrà che ne ha anche uno di patch.

[~/bbTutorial/build]

bitbake -c listtasks first  
Parsing recipes: 100% ...  
....  
do\_showdata  
do\_build  
do\_listtasks  
do\_patch  
...

Lanciando bitbake first verranno eseguiti entrambi i task, la patch e la build.

|  |  |
| --- | --- |
|  | Volendo, ora si può creare una ricetta che utilizzi la classe confbuild e un file append per eseguire le attività di patch, configurazione e build. |

### 8.3. I file include

BitBake ha due direttive per includere i file.

* include nomefile: è un inclusione opzionale, se nomefile non si trova nessun errore viene generato.
* require nomefile: se nomefile non viene trovato, viene generato un errore.

Vale la pena ricordare che il nome del file per "include" e "require" è relativo a qualsiasi directory in BBPATH.

#### 8.3.1. Aggiunta di un local.conf per l'inclusione

Un uso comune nei progetti BitBake è che bitbake.conf include un file local.conf che di solito viene posizionato nella directory di build.  
Il file local.conf può contenere delle impostazioni speciali per il target della build.  
Questa è la tipica configurazione di Yocto.

Se ne imita l'uso tipico di un local.conf. Ovvero, si fa in modo che bitbake.conf 'pretenda', con require, di trovare un file local.conf file.

Lo si fa aggiungendo le seguenti righe a meta-tutorial/conf/bitbake.conf.

meta-tutorial/conf/bitbake.conf

require local.conf  
include conf/might\_exist.conf

Eseguendo ora qualche build, BitBake mostrerà un lungo messaggio di errore dove l'ultima riga sarà qualcosa del genere:

ERROR: Unable to parse conf/bitbake.conf: …​Could not include required file local.conf

L'aggiunta di un file local.conf, anche vuoto, nella directory *build*, risolverà il problema.

Si noti che non c'è stata alcuna segnalazione per l'inesistenza del file conf/might\_exist.conf. Da tenere sempre presente che, al contrario di require, include ignora i file inesistenti.

|  |  |
| --- | --- |
|  | Tutto il codice di esempio è disponibile su <https://bitbucket.org/a4z/bitbakeguide> L'esempio per questo capitolo è ch08. |

## 9. Uso delle Variabili

La possibilità di definire variabili e usarle nelle ricette rende BitBake molto flessibile.

Le ricette possono essere scritte in modo che le parti configurabili utilizzino delle variabili.  
L'utente di tali ricette può [ri]definire i valori delle variabili che verranno poi utilizzati dalle ricette.  
Un tipico esempio è quello di passare una configurazione aggiuntiva o creare flag per una ricetta.  
Usando correttamente le variabili, non c'è bisogno di modificare o sostituire una ricetta solo perché in alcune funzioni c'è bisogno di argomenti speciali.

### 9.1. Variabili globali

Le variabili globali possono essere impostate dall'utente e le ricette esistenti le possono utilizzare.

#### 9.1.1. Definizione delle variabili globali

Un "local.conf" vuoto, come è stato fatto, non è molto utile. Quindi vi si aggiunge qualche variabile. Supponendo di stare nella directory di build si può eseguire:

[~/bbTutorial/build]

echo 'MYVAR="hello from MYVAR"' > local.conf

oppure usare un qualsiasi editor per aggiungerlo al file.

#### 9.1.2. Accesso alle variabili globali

Si può accedere a MYVAR nelle ricette e nelle classi. A scopo dimostrativo, si crea il nuovo gruppo di ricette recipes-vars con e una ricetta con dentro myvar.

meta-two/recipes-vars/myvar/myvar\_0.1.bb

DESCRIPTION = "Show access to global MYVAR"  
PR = "r1"  
  
do\_build(){  
 echo "myvar\_sh: ${MYVAR}" (1)  
}  
  
python do\_myvar\_py () {  
 print ("myvar\_py:" + d.getVar('MYVAR', True)) (2)  
}  
  
addtask myvar\_py before do\_build

|  |  |
| --- | --- |
| **1** | Accesso alla variabile con una sintassi simile a quella di bash. |
| **2** | Accesso alla variabile tramite il "global data store" [archivio dati globali]. |

Se ora si esegue bitbake myvar e se ne controlla l'output del log nella cartella tmp, si vedrà che si ha effettivamente accesso alla variabile globale MYVAR. Per il file di log, si deve cercare qualcosa di simile a questo: build/tmp/work/myvar-0.1-r1/temp/log.do\_myvar\_py.

### 9.2. Variabili locali

Una tipica ricetta principalmente fa riferimento solo a variabili utilizzate per impostare funzioni definite in classi che la ricetta eredita.

Per avere un'idea di come funziona si crea

meta-two/classes/varbuild.bbclass

varbuild\_do\_build () {  
 echo "build with args: ${BUILDARGS}"  
}  
  
addtask build  
  
EXPORT\_FUNCTIONS do\_build

e si usa tale classe nella ricetta

meta-two/recipes-vars/varbuild/varbuild\_0.1.bb

DESCRIPTION = "Demonstrate variable usage \  
 for setting up a class task"  
PR = "r1"  
  
BUILDARGS = "my build arguments"  
  
inherit varbuild

L'esecuzione di bitbake varbuild produrrà file di log che mostrano che la build rispetta il valore della variabile impostato nella ricetta.

tmp/work/varbuild-0.1-r1/temp/log.do\_build

DEBUG: Executing shell function do\_build  
build with args: my build arguments  
DEBUG: Shell function do\_build finished

Questo è un modo molto tipico di usare BitBake. Il task generale viene definito in una classe, come ad esempio download source, configure, make e altri, e la ricetta imposta le variabili necessarie per il task.

|  |  |
| --- | --- |
|  | Tutto il codice di esempio è disponibile su <https://bitbucket.org/a4z/bitbakeguide> L'esempio per questo capitolo è ch09. |

## 10. Sommario

Questo è tutto per questo tutorial, sono felice che di essere arrivato fino a qui e spero che sia piaciuto.  
Dopo aver letto questo tutorial si dovrebbe avere una solida base sui concetti di base su cui si basa BitBake.

Gli argomenti trattati sono:

* BitBake come motore che esegue script Python e/o Shell.
* Il layout solito del progetto BitBake e i path di default dei file.
* I fondamenti sui layer e le relazioni tra di essi.
* I 5 tipi di file utilizzati da BitBake (bb- bbclass- bbappend- conf- e include).
* Le funzioni e i task di BitBake, come organizzarle, raggrupparle e chiamarle.
* Le variabili di BitBake e il loro utilizzo di base.

Si spera che l'avere familiarità con questi argomenti sia d'aiuto per chi comincia ad utilizzare un progetto come Yocto e si chiede cosa stia succedendo.

Inserite pure un link al proprio progetto BitBake nella seguente parte dei commenti.